DATAPOINT

/*

The DASL Document

February, 1983

For Internal Use Only

The DASL Document:

A Tutorial Introduction & Programming Guide

February 8, 1983

Stuart Greene
Gene Hughes

CONFIDENTIAL PROPRIETARY INFORMATION

This item is the property of Datapoint Corporation, San Antonio, Texas, and contains confidential and trade secret information. This item may not be transferred from the custody or control of Datapoint except as authorized by Datapoint and then only by way of loan for limited purposes. It must not be reproduced in whole or in part and must be returned to Datapoint upon request and in all events upon completion of the purpose of the loan.

Neither this item nor the information it contains may be used or disclosed to persons not having a need for such use or disclosure consistent with the purpose of the loan, without the prior written consent of Datapoint.

0. PREFACE

DASL (pronounced "dazzle") is Datapoint's Advanced Systems Language. Some of you may have known DASL in the days when it was called BABEL, and this volume was called The BABEL Book. The name was changed when a trademark conflict was discovered.

This document is an attempt to familiarize programmers with DASL. It is intended to convey the "flavor" of programming in DASL, but is by no means a rigorous reference. For those occasions when precise language definitions are required, the reader may consult the DASL Language Reference Guide in the appendix.

The tutorial will be divided into three parts, as follows:

1. Discussion of concepts
2. Sample programs
3. Technical appendices

The DASL compiler is only available under RMS. While DASL itself is not operating system or machine dependent, the I/O packages currently available only support RMS operation. Therefore, readers who wish to type in and run the sample programs provided will have to do so under RMS.

The I/O package referenced in this tutorial is called SIO (short for "Sample Input/Output"). SIO and its compile time support file (SIOINC) will be made available, but readers are urged to note that these entities are not defined by the DASL language itself.

DASL was designed by Gene Hughes at Datapoint's Advanced Product Development. In many ways, DASL can be seen as the conceptual offspring of C, the systems programming language developed by Bell Laboratories. Readers familiar with C should feel right at home. The code generation component of DASL for both the 5500 and 6600 instruction sets was written by Mark Miller, also at APD.

We think that once the concepts of programming in DASL become clear, the consistency of syntax, simplicity of constructs and rich set of operators will make learning and using the full power of the language an enjoyable experience.

I. Be DASL'ed

Datapoint has a high-level systems programming language. Its name is DASL.

DASL is a "high-level" language in the sense that it provides the kinds of powerful and general constructs required for successful structured programming. On the other hand, DASL is one of the few high-level languages which does not "hide" any part of the machine from the programmer. Many tasks which previously demanded the "low-level" capabilities of an assembler can now be handled cleanly and elegantly in DASL.

A major aspect of DASL's design philosophy has been to achieve this power and generality through simplicity: DASL is easy to learn. DASL programs are easy to read, to maintain and to modify. DASL compiles quickly and produces machine efficient code.

This section of the tutorial will present some working DASL programs and their explanations. This tutorial is not aimed at breaking down the conceptual barriers of the computer-shy novice, but at giving the experienced programmer quick hands-on exposure.

A few comments before we begin. The DASL language itself does not contain any I/O facilities. Therefore DASL is not tied to any operating system or machine. Users may wish to write their own specialized I/O functions for whatever operating systems and whatever devices they require. There is a package named SIO, which provides some basic screen and keyboard I/O for an RMS workstation. SIO was written by Gene Hughes specifically for this tutorial. It allows a DASL program to read data from the keyboard and write data to the screen. Feel free to use SIO, but once you are comfortable with DASL, you are encouraged to scout out the other, more complex I/O packages that have been developed, or to "roll your own."

In the sample programs you will also notice directives to INCLUDE files with names like DINC, DRMS and SIOINC. These are for the most part just definitions required by the I/O package. The complete text of these files will be included in the appendix, along with source listings for SIO. Now, on to a real live DASL program.

II. A Real Live DASL Program

```
/* This program prompts the user for a number, then
computes and displays the factorial of that number */

/* The factorial of n is the value obtained by
multiplying together all of the numbers from 1 to n, for
example: the factorial of 4 is (1 x 2 x 3 x 4) = 24.
*/


INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(SIOINC)



FACTORIAL(N INT) INT :=
VAR MULT INT;
{
   MULT := 1;
   RESULT := 1;
   LOOP { WHILE MULT <= N;
      RESULT := RESULT * MULT;
      MULT := MULT + 1;
      };
   };




ENTRY MAIN() :=
VAR KEYVAL, NFACT INT;
{
   D$WRITE(SCREEN, LN, S,'Enter number: ');
   KEYVAL := D$READI(KEYBD);
   NFACT := FACTORIAL(KEYVAL);
   D$WRITE(SCREEN, LN, S,'Factorial: ', I,NFACT, LN);
   };
```

COMMENTS: In DASL, comments start with a /* and end with an */. Lines which begin with a period, plus sign or asterisk will also be treated as comments. Comments can be nested.

INCLUDE: Each include is a compile time directive to temporarily switch the current input file to the one indicated. Each of the three INCLUDEd files appears in the appendix.

PROGRAM FUNCTION: **FACTORIAL**

INPUT PARAMETER LIST: FACTORIAL is called with one parameter whose name in this case is N. N is a local variable of FACTORIAL whose type is INT, which is short for integer.

RESULT: The function returns a value of type INT upon completion. This value is also a local variable of the function, and is automatically named RESULT by the compiler.

VAR: This keyword indicates that local variable definitions follow. In this case, only one local variable (aside from N, the input parameter, and RESULT, the return value) is being defined. Its name is MULT and it is of type INT.

FUNCTION LOGIC: The body of the function between the outermost set of braces actually performs the operation of evaluating the factorial of N. When the processing terminates, the value left in the variable named RESULT will be returned to the caller.

PROGRAM FUNCTION: **MAIN**

Every DASL program has one function named MAIN, and it is where execution of the program starts. MAIN takes no parameters and returns no value. MAIN is always preceded with the keyword: ENTRY. This means that MAIN can be seen by entities outside of the file. We'll discuss import/export considerations later on.

FUNCTION LOGIC: MAIN is performing a series of function calls. D$WRITE and D$READI are both external functions defined by the I/O package. D$WRITE is used to display information to the screen. D$READI returns an integer value from the keyboard. The only substantial act performed by MAIN is the call to the FACTORIAL function, assigning the value returned into the local variable called NFACT.

The first thing to notice about the program is that it is made up of two blocks of code (FACTORIAL and MAIN) referred to as functions. One function in every DASL program is called MAIN, and it is where execution of the program starts. A function may or may not be called with parameters, and it may or may not return a value. DASL functions are more or less analogous with standard mathematical functions. For example:

$$f(x) = x^2 - 9$$

tells us the rule associated with changing an "input parameter" ("x") into the value returned by the function ("f(x)"). If this function is "called" with a value of x equal to 5, it will "return" the value of 16. The only substantial difference between this formal sort of function and what we'll be seeing in DASL is that mathematical functions deal only with values, while DASL functions deal with both values and effects. An effect is some change in the state of the world. For instance, I can call a DASL function which does nothing but change what the user sees on his screen. The function has propagated a very real effect, but its action has nothing to do with returning a value to the caller.

Functions consist of local variables and statements. Local variables are data elements which are the private property of the function which defines them.

Statements specify the **actions** which the function must take. Statements are composed of expressions and flow of control constructs.

Expressions are pieces of DASL code which have a value. For instance, (X + 2) is an expression. Statements, on the other hand, do not have values, only effects. The following is an example of a statement:

```
NUMVAR := (X + 2);
```

In this example, there are five different symbol groups which can be thought of as expressions:

NUMVAR, **X** and **2** are each autonomous expressions. **(X + 2)** is an expression, and **NUMVAR := (X + 2)** is as well. Each of these symbol groups can be thought of as having a value. In the case of NUMVAR := (X + 2), the expression has an effect as well as a value. By putting a semicolon after the whole expression, we turn the line into a statement, which causes only the **effects** to remain. The values associated with the expressions "go away". They are ephemeral, evaluated when necessary, but not stored in memory.

Note that the assignment operator in DASL is a colon followed by an equals sign (:=). It is usually pronounced as "gets" when reading a program aloud. The above example would read: "NUMVAR gets X plus two". Compound statements can be made by enclosing a group of smaller statements between open and close braces. The compound statement must also terminate with a semicolon. For example:

```
{
   X := OFFSET(N);
   NUMVAR := (X + 2);
   D$WRITE(SCREEN, LN, I,NUMVAR);
   };
```

This is a single DASL statement, which is made up of three component statements. In every way, the compound can be thought of as a single statement.

Even though execution always begins with MAIN, it is typically the last thing defined in the program module. This is because in DASL, every entity must be declared before it is referenced. Since MAIN will call the FACTORIAL function, FACTORIAL must appear first. When reading through a program, it is generally best to look at MAIN first and let the flow of the program logic guide you.

Notice that MAIN itself is not doing a great deal. It is simply putting up a prompt on the screen, calling a function to get an integer value from the keyboard, handing off that value to another program function (whose internal details MAIN knows nothing at all about) and displaying the value returned by that function call. That's it. MAIN's job is usually that of a traffic manager, defining the superstructure of a program, and calling special purpose functions to actually do the work.

In fact, the only thing that one function ever "knows" about any other function is its type, that is, what types of parameters (if any) it needs to be called with, and what type of value (if any) it returns upon completion. This is very nearly the complete interface between program functions, and it helps insure that only a function's behavior, and not its methodology, determines how a program executes. This is perhaps the single most important benefit of modular programming, because it bundles complexity into small and highly localized packages, which can be coded, debugged, re-implemented or whatever--independently of each other.

DASL provides several flow of control constructs, three of which are used in this program. The first is the most basic of all, the semicolon. Ending a statement causes the next statement, if any, to be executed. Flow of control doesn't get any simpler than that. When there are no statements left to execute, the function returns. If the function has been defined as returning a value, the contents of the variable named RESULT will be what is returned. When MAIN runs out of statements to execute, the program terminates.

The next flow of control construct used is the function call. The statement which reads:

```
NFACT := FACTORIAL(KEYVAL);
```

calls the FACTORIAL function with KEYVAL as its parameter, and assigns the value returned into the variable NFACT.

The last flow of control construct used in this program is seen inside the body of FACTORIAL. It is the LOOP/WHILE construct and true to its name, causes the statement following the LOOP to be continually re-executed as long as the expression following the WHILE remains logically true, that is, evaluates to a non-zero value. As soon as the expression becomes logically false, that is, takes on a value of zero, the loop is exited. Remember that compound statements can be made by putting braces around a group of smaller statements. The statement which is being LOOPed here is such a compound, since it is composed of three smaller statements.

It is interesting to note that even though the WHILE statement in this example was the first one within the body of the LOOP, this does not have to be the case. Not only that, but a LOOP may have several different WHILE statements, providing for multiple exits from the LOOP based on different conditions. And of course, any LOOP which doesn't have at least one WHILE statement in it will run forever. Or until you do something dramatic:

```
SARTRE() :=
{
   LOOP {
      D$WRITE(SCREEN, LN, S,'HELP! NO EXIT!');
      };
   };
```

The rest of the factorial program should be rather familiar stuff: expressions, assignments, various operators. In the interest of clarity the first sample program took no stylistic shortcuts. But DASL does provide a number of notational conveniences for the programmer which are frequently used. Below, I have rewritten the FACTORIAL function, using several of these conventions:

```
FACTORIAL(N INT) INT :=
VAR MULT INT;
{
   MULT := (RESULT := 1);   /* Parentheses optional */
   LOOP { WHILE MULT <= N;
      RESULT *= MULT++;
      };
   };
```

There are three shortcuts being used in this example:

First, the two assignment statements before the loop have been merged into one statement. RESULT gets the value of 1, and MULT gets the value of the expression (RESULT := 1). The value of an assignment expression is the value being assigned. The assignment expression has the side effect of assigning its value into some other data element. Remember, expressions must have values and can also have effects. Statements can only have effects.

Next, notice the ++ following MULT. This is DASL shorthand for increment. If the ++ (or -- for decrement) comes before the variable name, then the incrementing is done before the value is used in the expression. If, as in the above example, the operator comes after the variable, the current value is used in the computation, then the value is incremented. If a variable is used more than once in an expression, the order of evaluation is undefined and the increment/decrement operators should not be used.

The last trick is the *= operator. Very often, we need to modify the value of a variable in some way, then plug the modified value back into the source variable. This is what we did when we said:

```
RESULT := RESULT * MULT;
```

The *= notation means the same thing, not only saving keystrokes but emphasizing the logical relationship between the left and right sides of the statement. Every binary arithmetic and bitwise operator in DASL can be used in this way (see the Language Reference Guide in the appendix for a complete list of DASL operators).

Before going on, it would probably be helpful for you to get the sample program up and running (so you can convince yourself that the language really works!). Turn the page for instant enlightenment, or a reasonable facsimile thereof.

III. The Joy of Running

Now that you've got a real live DASL program, the next step is to get it up and running. The path from DASL source to executable object feels like a single operation, but in fact, the process has several components. For that reason, most DASL programmers use a chain file to handle the less than fascinating details. After all, that's what machines are for. The following chain, which I call COMPLINK (for **COMPILE** and **LINK**), will work just fine for programs which use SIO as their I/O package and have only a MAIN module. Feel free to use it, and to add improvements as you go along.

```
DASL #IN#
SNAP DASLASM,#IN#;N
LINK ,#IN#/CMD;ERR,FASTLIB,NEW
SEGMENT #IN#
INCLUDE #IN#
INCLUDE D$LIB.D$START
LIBRARY SIO
LIBRARY D$LIB
LIBRARY RMSUFRS
SIGNON #IN#
*
```

Assuming that you have named your program something inspired like **FACTORIALIZE**, this is what your command line will look like:

```
COMPLINK/CHN;IN=FACTORIALIZE
```

COMPLINK will take care of the details of compilation and linkage, and leave behind, assuming no errors are encountered, an executable file called FACTORIALIZE/CMD. By the way, if the compiler does turn up any errors, it will create a file called DASLERR/TEXT. DASLERR is a standard text format file which will describe each of the errors encountered and tell you the line number in your source where the offense occurred.

Taking a closer look at COMPLINK, you'll notice that it really consists of three parts. First, there is a call to the DASL compiler, which turns your DASL code into assembler source. Next, there's a call to SNAP, which generates a relocatable code module. The last step is an invocation of LINK, which performs the final blending of modules required to make an executable command file.

Following the line in which LINK is called, there is a stack of directives which specify to the linker which modules are required to resolve all of your program's references. The file called D$LIB contains the run-time support needed by DASL programs. One of D$LIB's members, D$START, is explicitly INCLUDEd in the link because it is responsible for starting up the execution of a DASL program. In effect, it forges an interface between the command interpreter and the program. The rest of the run-time package contains the functions necessary for recursion support, software divide and multiply for the 5500, and block move/comparison. All things considered, that's not a whole lot of run-time. SIO, of course, is the Sample I/O package. RMSUFRS is the standard RMS package of User Function Routines. Some of these get called by SIO, and must therefore be included.

If your program consists of more than one module, or references other function libraries, simply add them to the list. The linker's INCLUDE directive will take the entire /REL module specified, and plug it into the /CMD file, merrily resolving references along the way. If you use the LIBRARY directive instead, the /REL file will only be searched for those entities required to resolve references. There is a separate manual for the LINK command which will fill you in on all of the details.

The DASL compiler itself consists of two serial passes, one which transforms DASL source code into a machine independent, intermediate logical representation, and another which generates the actual output code. Because pass one is machine independent, different code generators may be hooked onto it for different machines. At present, the default code generator produces SNAP assembly source for the 6600 instruction set. The default output file name is DASLASM.

If you want to **run** your programs on a machine which uses the 5500 instruction set (5500s and 3800s), you will have to change the line which calls the DASL compiler so that it looks like this:

```
DASL #IN#;CODE=5500
```

The compiler itself must be run on a machine which uses the 6600 instruction set. For those of you who left your scorecards at home, at present that means a 6600, an 8600 or an 8800. And if you forget what the options and defaults are, you can always type:

```
DASL;HELP
```

The existing compiler package, DASL/CMD, stops after producing assembler source code. It is the responsibility of the rest of COMPLINK to SNAP and LINK the output to produce the executable command file. In the future, some or all of these services may be provided by the compiler package. In the meantime, you can expect to see all sorts of handy little CHAIN files hanging around your work catalog. As time goes on, programming accessories including a symbolic debugger should become available, making life a little bit easier and a little more productive for one and all.

IV. Anatomy of a DASL Program

The standard model of computing hasn't changed much since John von Neumann came up with the idea of the "stored program machine." Basically, the computer's memory is seen as a bucket into which we pour a molten stream of information. Some of this information is supposed to be interpreted as program code, the stuff which gets executed and which controls what goes on. The rest of the information is taken to be data elements which get manipulated by the various actions, or effects, of the program instructions. Very neat, very simple, and imperative that whoever is in control of all of this keeps the relationship straight. Executing your data can have some rather amusing results.

Given that the program ultimately translates into organized chunks of memory, it is the job of the high-level language to insure that the information which gets deposited into memory is not only managed correctly, but is a logical reflection of what the programmer has in mind. That is why constructs such as arrays, integers and pointers were invented. To provide the programmer with "templates" for the systematic and meaningful organization of his program's memory space.

The largest modular unit in DASL is the PROGRAM. A PROGRAM is made up of one or more MODULES, exactly one of which is called the MAIN MODULE, because it contains the function named MAIN, where program execution starts. Some programs will consist only of a MAIN MODULE. One MODULE corresponds to exactly one source file, and each module is independently compiled into a relocatable code file. Programs which consist of more than one MODULE will need to link the component /REL files together, as discussed in the preceding chapter.

Individual MODULES are made up of definitions and declarations of the actual entities which will inhabit memory when the finished program is executed. It is helpful to think of these entities as being of two categories: FUNCTIONS and DATA VARIABLES.

FUNCTIONS are autonomous program routines dedicated to carrying out a specific task. Functions are made up of statements and expressions.

DATA VARIABLES are named pieces of memory of various types. Variables may range from being globally visible to an entire program down to being the private property of a single statement within a function.

Both FUNCTIONS and DATA VARIABLES have type information associated with them, and the compiler does type compatibility analysis to help "keep the programmer honest." As we discuss the different data types and their properties, keep in mind that DASL also provides a facility, called "casting" which allows local reinterpretation of any data element's type. With casting, the programmer is never unduly restricted by someone else's conception of how the machine and its memory should be treated.

DASL provides for five different classes of data. Each of these classes contains specific types. On the following pages we will examine each of the data classes, the types associated with them, and look at some examples of how they would be declared and used in a DASL program.

SCALAR - A scalar describes an integer value. A scalar can be of type:

BOOLEAN	- One byte, unsigned
CHAR	- One byte, unsigned
BYTE	- One byte, unsigned
UNSIGNED	- Two bytes, unsigned
INT	- Two bytes, signed
LONG	- Four bytes, signed

Example:

```
ENTRY MAIN() :=
VAR CHAR1, CHAR2 CHAR;
    NUM1, NUM2 INT;
    COUNTER UNSIGNED;
{
   CHAR1 := 'A';        /* Assigns value of ASCII 'A' to CHAR1 */
   CHAR2 := 231;               /* Assigns decimal 231 to CHAR2 */
   CHAR1 := CHAR1 + '*';       /* Adds '*' to previous val */

/* Remember, CHARs are just small, unsigned integers  */

   NUM1 := CHAR1 + CHAR2;
   NUM2 := NUM1 - (012 + CHAR1); /* 012 is an octal constant */
   NUM1++;                       /* Increment NUM1 */
   NUM2--;                       /* Decrement NUM2 */

   COUNTER := 64000;  /* Set up a 64000 step decrement loop */
   LOOP WHILE COUNTER-- > 0;
   };
```

ARRAY - An array is a continuous group of some type of data element. The data elements which make up the array can be of any defined data type, except for FUNCTION. The elements can themselves be arrays, giving the effect of multi-dimensionality with a single primitive class. The elements of an array are numbered from zero to one less than the number of elements in the array, so the first element of an array would look something like: ARRAYNAME[0].

Example:

```
ENTRY MAIN() :=
VAR TEXTLINE [15] CHAR;        /* Array of 15 characters */
    PAGE [24][15] CHAR;        /* Array of 24 arrays of 15 chars */
    NUMLINE [10] INT;          /* Array of 10 integers */
{
   TEXTLINE[0] := 'A';
/* Assign an ASCII A into the first position of textline */

   PAGE[0] := TEXTLINE;
/* Assign the TEXTLINE array into the first element of PAGE */

   PAGE[1][5] := TEXTLINE[0];
/* Assign the first character of TEXTLINE into the sixth character
of PAGE's second array */

   D$WRITE(SCREEN, C,PAGE[1][5]);
/* Write the character from the previous example to the screen */

   NUMLINE[0] := 10;
/* Assign a decimal value 10 into the first element of NUMLINE */

   NUMLINE[1] := NUMLINE[0] + TEXTLINE[5];
/* Assign the sum of the first element of NUMLINE and the 6th
element of TEXTLINE into the 2nd element of NUMLINE */
   };
```

POINTER - A pointer is a data element which contains the address of some entity in memory. The two primitive pointer operations which are available in DASL are:

& - "address of" operator. Takes the address of the entity following it. Example: &CHARLINE[0] would represent the address of the first element of the array named CHARLINE.

^ - "indirection" operator. Takes the value of what the pointer is pointing at. Example: CHARPOINT^ would evaluate to the actual entity in memory that CHARPOINT is holding the address of.

The type of a pointer is defined solely by the type of the thing to which it points. All pointers themselves are the same size, that is, the size of the address word of the machine on which the program is to be run. For the current Datapoint architecture, this magic number turns out to be two bytes.

Example:

```
ENTRY MAIN() :=
VAR CP ^ CHAR;
    C CHAR;
    ARRAY [10] CHAR;
{
   ARRAY := 'ABCDEFGHIJ';    /* Initialize the array with chars */
   CP := &ARRAY[4];   /* CP gets the address of the 5th char */
   C := CP^;          /* C get the character pointed to by CP */
   };
```

The net effect of these operations is equivalent to:

```
                  C := ARRAY[4];
```

That is, C winds up containing the ASCII 'E' from the array. In the sample programs later on, we'll be seeing a large number of non-trivial pointer operations.

STRUCTURE/UNION - STRUCTURES and UNIONS are similar classes of data. In a structure, the programmer can define a compound type made up of any of the available data types, except for FUNCTION. For instance, if you were building a linked list, you might want a structure that contained an integer value followed by a pointer to the preceding structure in the list followed by a pointer to the next structure in the list. Each of these three fields can be named and referenced, and the entire structure can also be manipulated as a logical whole.

The structure allocates sequential memory locations for the separate fields defined within it. The union is also composed of a group of subfields, but they all share a common starting point. The union is only intended to hold one value at a time, but the type of that value can be freely chosen from any of the data types of its members. Unions are usually used within structures, to represent an element whose type is different under different run time conditions. The amount of memory allocated for the union is the amount of memory needed for its largest member.

Example:

```
ENTRY MAIN() :=
VAR APPOINTMENT STRUCT {                /* Struct of 4 data fields */
       MONTH, DAY, YEAR INT;
       REMINDER [25] CHAR;
       };
{
   APPOINTMENT.MONTH := 06;
   APPOINTMENT.DAY := 23;
   APPOINTMENT.YEAR := 82;
   APPOINTMENT.REMINDER := 'NATIONAL SOFTWARE DAY    ';
   };
```

Note that the way we refer to a particular field of a structure (or a union) is: struct.field. Structures can have fields which are themselves structures. The hierarchical field naming convention simply continues, as in:

```
                  struct1.struct2.field
```

since struct2 is simply a field of struct1 which happens to also be a structure.

TYPDEF - TYPDEF is not itself a data class, but rather, is a facility whereby the programmer can invent and name his own data types. For instance, in the STRUCTURE example above we saw a local variable called APPOINTMENT which was a structure of 4 fields, named MONTH, DAY, YEAR and REMINDER. If we had first used the TYPDEF facility to create a data type called DATE, we could have defined APPOINTMENT as being a data element of type DATE. Or we could have made arrays of DATEs or included DATEs as fields of other structures, and so on. Here's what the TYPDEF and variable declaration portion would look like:

```
TYPDEF DATE STRUCT {          /* Define new data type: DATE */
         MONTH, DAY, YEAR INT;
         REMINDER [25] CHAR;
         };


APPOINTMENT DATE;
DATEARRAY [50] DATE;
DATELINK STRUCT {            /* Define a new structure, made  */
   ADATE DATE;               /* up of a DATE, followed by a   */
   NEXTDATE ∧ DATE;          /* pointer to DATE, for a linked */
   };                        /* list, etc.                    */
```

FUNCTION - It turns out that the program functions themselves are really another type of variable! After all, they live in memory. Their values are determined by the actual pattern of machine codes associated with them, their types by the types of their input parameters and return values. And while it is probably not desirable to change the function's value at run time (it is possible!), there are other powerful things which we can do because of this interpretation. For instance, we can store the address of a function into a pointer and later call the function indirectly. We can build arrays of pointers to functions and gain indirect logical control over their patterns of execution. All of this is a logical outgrowth of DASL's high-level/low-level philosophy.

Because of DASL's type compatibility restrictions, two entities which are "coincidentally," and not formally, of the same apparent type are not taken to be compatible (see the Language Reference Guide for complete details). For this reason, if you really did want to call a function indirectly, it would be necessary to first use the TYPDEF facility to create the definition of a function type, and then declare both the actual function and pointers to it to be of this type. Let's look at our factorial program in this light:

```
TYPDEF FTYPE (N INT) INT;  /* Define a function type: FTYPE */

FACTORIAL FTYPE :=         /* Function's type defined as FTYPE */
VAR MULT INT;
{
   MULT := RESULT := 1;
   LOOP { WHILE MULT <= N;
      RESULT *= MULT++;
      };
   };

ENTRY MAIN() :=
VAR KEYVAL, NFACT INT;
    FUNCPTR ^ FTYPE;   /* Pointer to a variable of type FTYPE */
{
   D$WRITE(SCREEN, LN, S,'Enter number: ');
   KEYVAL := D$READI(KEYBD);
   FUNCPTR := &FACTORIAL; /* FUNCPTR gets FACTORIAL's address */
   NFACT := FUNCPTR^(KEYVAL);  /* Indirect function call */
   D$WRITE(SCREEN, LN, S,'Factorial: ', I,NFACT, LN);
   };
```

This program is identical to our original example, except that FACTORIAL is being called indirectly (see program comments).

The CAST Operator - As with the TYPDEF facility, CAST is not itself a data class or type. The cast operator allows the type definition of some entity to be locally reinterpreted, dissolving type incompatibilities and permitting operations (hopefully under the programmer's close scrutiny) which the compiler would ordinarily consider to be in error.

DASL is not a "safe" language. It is possible, for instance, to over or under subscript an array. This would probably be an unacceptable condition for an applications oriented language, but DASL's responsibility is to provide the most powerful programming tools it can, within the context of high-level convenience and modularity. The cast operator permits the programmer to view memory as a pure pattern of bytes, independent of any particular, external interpretation. It is up to the programmer, when casting, to keep the world in order.

The cast operator appears as a set of angle brackets (the less than, greater than symbols) around a type specification. The data element following the cast operator is, in the local context, reinterpreted as being of the type shown between the brackets. For example:

```
        P ^ CHAR;
        I INT;

        {
           I := I + <INT>P;
           };
```

In this example, P, which is defined as a pointer to a character in memory, is being locally re-cast as an integer. Therefore, the pure 16 bit value which resides in P will be temporarily treated as an INT (signed 2-byte integer) for the operation specified. P is unchanged by the process; only its interpretation in the local context has been altered.

Casting is an exception rather than a rule. Very few programs need to cast extensively. The DASL compiler, which is probably the most complex program yet written in DASL, only resorts to extensive casting when using its dynamic storage allocator, D$ALLOC. In that instance, the motivation for casting is clear: to the storage allocator, memory is just memory. It is the outside world which wishes to impose high level structure information onto it. This duality forces D$ALLOC to resort to casting in order to manage its increasingly more complex interpretation of memory. When we call D$ALLOC, we must cast a pointer to the structure we wish to allocate as a pointer to CHAR. That's sufficient to get the starting address of the allocated memory into our pointer variable. Subsequently, the pointer can be used as we intended, reverting to its originally defined type interpretation. The following example illustrates a typical use of D$ALLOC function:

```
DEFINE(FILESIZE,30)

TYPDEF ITEM STRUCT {
         FLAG CHAR;
         FILENAME [FILESIZE] CHAR;
         ACTION CHAR;
         NEXT ∧ ITEM;
         };

ITEMPTR ∧ ITEM;

{
   < ∧ CHAR> ITEMPTR := D$ALLOC(SIZEOF ITEMPTR∧);
   };
```

Casting is a powerful technique, best used sparingly, when no other approach will suffice.

V. Sorting It Out With Pointers

With all of that theory under your belt, you're probably itching for some more code. The following program will perform a bubble sort on an array of characters entered from the keyboard, and scroll the intermediate results on the screen. The program will perform its particular magic using pointers. Even though the array to be sorted is a private, local variable of MAIN, other functions can be given selective access to it through a pointer which contains its address. This technique minimizes the amount of data which needs to be passed back and forth between functions while preserving the careful control of modularity afforded by the local variable concept. You will also see some simple pointer arithmetic being performed, which is explained in the program annotations.

Right at the beginning of the program, after the standard INCLUDEs, you will notice a DEFINE directive. DEFINE allows the programmer to do text substitution (see MACROCHAPT for full details). The first text element between the parentheses will be replaced with the text of the second element (the one after the comma) when encountered anywhere in the program, except inside of comments and literal strings. In this case, by having ARRAYSIZE symbolically DEFINEd and then using that definition throughout the program, if the desired array length should change at a later date, we need only change the DEFINE directive and recompile. In general, DASL programmers avoid "magic numbers" in the body of their code with a ferocity equaled only by their loathing of GOTO statements.

This program also introduces the IF/THEN flow of control construct. If the expression following the IF evaluates to a logical TRUE, that is, if its value is anything but zero, the statement following the THEN clause will be executed. If the expression takes on a zero value, the rest of the statement is not executed. Remember that the statement following the THEN can be a compound statement block if you so choose. DASL also has an IF/THEN/ELSE construct. If the test expression is found to be FALSE, that is, equal to zero, the statement following the ELSE is executed, rather than just falling through as in the simpler IF/THEN construct.

The last new twist introduced in this program is the SIZEOF operator, which evaluates to the number of bytes belonging to the entity following it. For instance, if NUMARRAY were defined as an array of 10 integers, the expression (SIZEOF NUMARRAY) would evaluate to 20, since each integer is 2 bytes long. SIZEOF may be applied to all DASL data types except for function, whose size is ultimately machine dependent. Enough talk. Let's code.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(SIOINC)
DEFINE(ARRAYSIZE,80)

SWITCH(CP1, CP2 ∧ CHAR) :=
VAR TEMP CHAR;
{
   TEMP := CP1∧;
   CP1∧ := CP2∧;
   CP2∧ := TEMP;
   };


GETARRAY(P ∧ CHAR, SIZE INT) INT :=
{
   D$WRITE(SCREEN, LN, S,'ENTER ARRAY', LN);
   RESULT := D$READS(KEYBD, P,SIZE);
   };


RESPONSE() CHAR :=
{
   D$WRITE(SCREEN, LN, S,'ANOTHER SORT?    ', F);
   RESULT := D$READC(KEYBD);
   D$READC(KEYBD);  /* Bump past <ENTER> key */
   };


ENTRY MAIN() :=
VAR P, MAXPT ∧ CHAR;
    ARRAY [ARRAYSIZE] CHAR;
    LEN INT;
{
   LOOP {
      LEN := GETARRAY(&ARRAY[0], SIZEOF ARRAY);
      MAXPT := &ARRAY[--LEN];
      LOOP { WHILE --MAXPT >= &ARRAY[0];
         P := &ARRAY[0];
         LOOP {
            D$WRITE(SCREEN, SL,&ARRAY[0],LEN, LN);
          WHILE (P + 1) <= MAXPT;
            IF P∧ > (P + 1)∧ THEN SWITCH(P, P + 1);
            P++;
            };
         };
    WHILE RESPONSE() = 'Y';
      };
   };
```

The function called SWITCH takes a pointer to the two elements in the array which need to have their positions switched. CP1 points to the first of these characters, and the expression CP1^ represents the actual character to which CP points. GETARRAY prompts the user for the data to be sorted, returning the number of characters actually entered. RESPONSE will get called when the entire sorting process is finished, and will ask the user if he wishes to enter another array of data. If the character returned by RESPONSE is a 'Y', the major loop in MAIN will be re-executed. Any other character will cause the program to terminate.

As advertised, this program engaged in some pointer arithmetic. In MAIN, only one pointer into the array, P, is maintained. The next character in the array is handled implicitly, in terms of the expression (P + 1). Adding a scalar value to a pointer, as in the expression (P + 1), causes the pointer to point at the next object in memory of the same type. This is equivalent to increasing the value of the pointer by the size of the scalar multiplied by the size of the data element pointed to. If the object pointed to were, for instance, a 57 byte long structure, incrementing the pointer by 1 would increase the pointer's value by 57. Adding 2 would increase the value by 114 and so forth. In this case, since the object we are pointing at is only one byte long, adding 1 to the pointer only increases its value by 1. Subtracting a scalar from a pointer works in the same manner. Incidentally, the ++ and -- operators can also be used.

Take some time out to read the program carefully. Start with MAIN and follow the logic as it threads through the other program functions. Kind of looks like a bubble sort, doesn't it? Right now, the display statement is tucked inside of the inner loop, so we can see all of the intermediate states of the sort. You might like to play around with moving the position of the display statement. You might, for instance, only output the array when elements need to be switched. Feel free to "break" the program, deliberately introduce mistakes and observe the kinds of errors which the compiler comes up with. It's more fun than educational TV.

VI. Recursion: See Recursion

Another strange and powerful feature of DASL is the ability to use recursive functions, that is, functions which can call themselves. Recursion is a powerful technique for a number of programming applications, including sorts and various data structure manipulations (one subtree looks a whole lot like another from up here).

The factorial program we saw at the beginning of the tutorial is a prime candidate for recursion. After all, what is n factorial but n times (n - 1) factorial, and so on? In fact, there are only two things you ordinarily need to keep in mind when constructing recursive functions:

1. The function must have a "bottom", that is, a condition in which the functions stops calling itself and allows the stack to pop all of its accumulated return values....

2. Each call of the function must, in some sense, bring you closer to the "bottom", that is, the series of function calls must be guaranteed to "converge" on the non-recursive condition.

There's no need to rewrite the MAIN function of the factorial program, it stays just the same. The FACTORIAL function, written recursively, looks like this:

```
RECURSIVE FACTORIAL(N INT) INT :=
{
   IF N = 0 THEN RESULT := 1
      ELSE RESULT := N * FACTORIAL(N - 1);
   };
```

The keyword RECURSIVE precedes the function name, and informs the compiler that the function can call itself. This is important, since at compile time there is no way to determine how deep the recursion stack will get, and space for the function must be dynamically allocated. The beauty of recursion is that the programmer is free to think of the algorithm in totally relative terms. N factorial is expressed in terms of (N - 1) factorial. As long as there is a bottom, it will all get resolved in the end.

Another simple example of recursion can be found in the Fibonacci series. Each Fibonacci number is equal to the sum of the two preceding Fibonacci numbers. The series is usually started with the 0th value = 0 and the 1st value = 1. Using these two as the bottom conditions, you might try writing a recursive function to evaluate the n'th Fibonacci number.

VII. Random Observations on a STATIC Theme

In general, each time a function is called, all of its local variables are "recreated." Input parameters take on the values passed from the caller, and all other variables are of initially indeterminate value.

If, however, you wish to retain a value from one call to the next, the local variable declaration may be prefaced with the keyword STATIC. In the following example, a random number generating function has declared its seed variable to be STATIC. In this way, each time a new random number is requested, the last random value generated seeds the calculation. (Of course, these numbers are "pseudo-random", since they are generated by a deterministic process. To quote von Neumann, "Anyone who considers arithmetical methods of producing random digits is, of course, in a state of sin." Sorry John.)

```
DEFINE(K1, 31769)
DEFINE(K2, 28489)
DEFINE(K3, 32767)

RANDOM() UNSIGNED :=
VAR STATIC SEED UNSIGNED := 14723;
{
   RESULT := SEED := ((SEED * K1) + K2) % K3;
   };
```

The three "big numbers", K1, K2, and K3 are nothing special. In fact, for this algorithm to work at its best they should be "relatively prime." Perhaps some closet number theorist out there will write a DASL implementation of Euclid's Algorithm, so we can find some better values and generate some nastier random sequences.

Notice that the local variable, SEED, is STATIC. Each time RANDOM is called, SEED will start out with value left over from the previous call. Also note the initialization capability which we haven't seen before. STATIC and global variables (which are simply variables defined outside of a function) can be initialized when they are defined. SEED's initial value of 14723 will only be applied the first time RANDOM is called. If you want to re-initialize a variable each time a function is called, you must use a regular assignment statement in the body of your code, and not this initialization facility.

There is another new operator lurking in the program, the modulo operator. The modulo operator, represented by the percent sign ('%'), evaluates to the remainder after division when the expression to the left of the operator is divided by the value of the expression to the right of the operator. For instance, the value of **(14 % 3)** is 2, since 14 divided by 3 yields 4 (throw it away) with a remainder of 2 (that's the good part). So, all positive values taken **% 10** (pronounced "mod 10") are mapped into the range 0 to 9 and all negative values into the range -9 to 0.

By the way, now that you have a simple random number generator, you can write a cute program which displays a character to the screen at a random location. Let your horizontal position be determined by RANDOM() % 80 and your vertical by RANDOM() % 24 (of course, a self-respecting DASL programmer would use DEFINEd quantities such as SCREENWIDTH and SCREENHEIGHT instead of the raw numerical values, but let your conscience be your guide). You can even make the character displayed random by making it: (RANDOM() % ('~' - ' ')) + ' '. Have fun!

VIII. How To Succeed In Branching Without Really Trying

A familiar programming problem concerns the different actions which need to be taken depending upon the run time value of some expression. For instance, if we get a character from the keyboard, we may first wish to determine if it's a backspace, cancel, cursor up, and so forth, and take the appropriate action. If the character doesn't turn out to be one of our special control functions, we may wish to take a default action, for instance, just displaying the character.

DASL provides a multi-branch flow of control construct called the CASE statement, which simplifies this kind of programming task. In the following example, let's assume that we have used the DEFINE capability to symbolically pre-specify the values which would be received for the various keyboard control keys. A program function which intelligently echos keystrokes from the terminal might look something like this:

```
ECHO() :=
VAR IN CHAR;
{
  LOOP {
    IN := D$READC(KEYBD)      /* Get character from keyboard */
   WHILE IN ~= ENDCHAR;       /* Stay in LOOP while not end */
    CASE IN {                 /* CASE switch on value in IN */
      BCKSPACE: BACKSPACE();
      CANCL: CANCEL();
      TABCHAR: TAB();
      CURSUP: CURSORUP();
      CURSDN: CURSORDN();
      CURSRT: CURSORRT()
      CURSLF: CURSORLF();
      F1: FUNC();
      F2: FUNC2();
      F3: FUNC3();
      F4: FUNC4();
      F5: FUNC5();
      DEFAULT: DISPLAY(IN);   /* Defaults to displaying IN */
      };
    };
  };
```

Within the CASE statement, the case labels on the left side of each line are compile time constants, followed by a colon. If the value of the CASE expression (in this instance, the variable IN) matches the value of any of the case labels, the statement following the case label is executed. At the end of the case list, there is an optional case label called DEFAULT. The statement following it is executed if the switching expression does not match any of the other case labels. If no DEFAULT is specified and no match is found, then execution "falls through" the CASE statement.

DEFINE(MACROCHAPT, The DASL Macro Facility)

IX. MACROCHAPT

DASL contains a language within a language. It is the compile time macro facility, which is used to logically manipulate program text before the stage we normally think of as compilation. In other words, the macro facility controls the actual text which the compiler eventually sees.

In a strong sense, we never directly program in DASL. **We** always write our source in a kind of SUPERDASL, which gets macro processed into normal DASL source, ripe for compilation. And though it is never truly necessary to think of the division this radically, it does emphasize the fact that the macro facility defines another layer of language, syntactically distinct from the rest of DASL.

We have already seen two macro primitives at work: DEFINE and INCLUDE. DASL has three other pre-defined macros, namely: IFELSE, INCR and SUBSTR. Using these five primitives, it is possible to construct a wide variety of user macros which can simplify and clarify your programs. In fact (and this is a rather amusing result), the macro facility in DASL constitutes a Universal Turing Machine. In other words, every [finite] computational problem is theoretically solvable using **only** the macro language. Just thought you'd like to know.

Let's dispose of the INCLUDE directive right off the bat. INCLUDE simply changes the compiler's current input file. So in fact, the INCLUDE directive does not get replaced with any other text. It simply has the compile-time **effect** of temporarily changing the source file. As with any other programming environment, INCLUDE is useful for neatly packaging logically related sets of macros, declarations and the like. Take a look at the listings of DINC, DRMS and SIOINC in the appendix for some good examples.

DEFINE is used to create new macros. A new macro may be as simple as the one-to-one relationship of an identifier with a text string to be substituted for it. All of the cases of DEFINE we have seen so far have been of this variety. For example:

```
DEFINE(ARRAYSIZE,80)
```

has the effect of substituting an 80 every time the identifier ARRAYSIZE is encountered in the program. The DEFINE substitution is active everywhere in the program except for comments and literal character strings. In this simplest form, DEFINE is an aid to the symbolic management of a program's "magic numbers." But used to its fullest, the DEFINE directive allows you to create powerful compile-time programs which can simplify your coding effort. Let's start by looking at an extremely simple example:

```
DEFINE(ADD,((#1) + (#2)))
```

The example defines a new macro called ADD. The #1 and #2 represent the parameters which will be supplied each time the ADD macro is invoked. For instance, consider the following program statement:

```
NUMVAR := ADD(7,4);
```

The ADD macro takes the numbers 7 and 4 to be the values of parameters #1 and #2 respectively, and the compiler sees the statement expanded like this:

```
NUMVAR := ((7) + (4));
```

The macro invocation looks just like a function call, doesn't it? The difference is, of course, that in-line text substitution is taking place at compile-time, as opposed to parameter swapping taking place between functions at run-time.

There seem to be a lot of parentheses in the macro definition which in turn show up in the expanded text. These are extremely important! Take a look at the definition of ADD. The outer set of parentheses is required by the syntax of the DEFINE directive. So much for those. The next set ensures that the contents of the macro are insulated from the invoking context. Even though there will be many situations in which the evaluation precedence of the expanded text is unambiguous, the binding parentheses impose an explicit order of evaluation, preserving the desired logic. The same holds true for the binding parentheses around each of the individual parameters.

To prove the point, let's take a look at what might happen if these parentheses were omitted. Assume for a moment that we had defined ADD in the "obvious" way:

```
DEFINE(ADD, #1 + #2)
```

Now, let's suppose that we invoked ADD with some interesting parameters:

```
NUMVAR := 5 * ADD(3 = 4, 5 | 7);
```

First, what value did we want NUMVAR to get? Well, the value of the expression 3 = 4 is FALSE, that is, 0, and the value of the expression 5 | 7 (5 **logical or** 7) is TRUE, namely 1. So ADD should evaluate to 0 + 1 (everybody get 1?) and NUMVAR should get 5 times that, namely 5. So much for theory.

What really happens is that our "obvious" ADD macro expands the statement as follows:

```
NUMVAR := 5 * 3 = 4 + 5 | 7;
```

When it comes to implicit binding, DASL has other things in mind. Applying DASL's precedence rules (see the Language Reference Guide) this statement is equivalent to:

```
NUMVAR := ((5 * 3) = (4 + 5)) | 7;
```

Stepping through the evaluation, one comes to the dismal conclusion that NUMVAR gets 1. The proof is left up to the reader as an exercise, as all of those legitimate textbooks like to say. So the moral is, always insulate your macro definitions - parentheses are cheap.

Time for some more macro magic. Consider the following definitions:

```
DEFINE(NUMCOUNT,0)

DEFINE(NEXTNUM, #[
   NUMCOUNT
   DEFINE(#[NUMCOUNT#], INCR(NUMCOUNT))
   #])
```

NUMCOUNT represents the simple case of a DEFINE; the kind which we have seen before. The identifier NUMCOUNT just gets replaced with a 0 whenever it is encountered. The interesting part of the example is NEXTNUM, which uses NUMCOUNT as a "macro-time variable." More on this in a moment.

NEXTNUM is a curious little macro that has two parts. First of all, it will return the current value of NUMCOUNT, which we have initialized to 0. Next, it will have the effect of incrementing NUMCOUNT, so the next invocation of NEXTNUM will return, true to its name, the next number.

The important twist introduced here concerns the "protection brackets," the #[and #] symbols which enclose several pieces of the macro. These ensure that the enclosed text will not be expanded when the macro is scanned. The only effect of scanning a protected piece of text is the fact that the protection brackets are stripped away. Keeping this in mind, let's look at what the NEXTNUM macro expands into when invoked:

```
NUMCOUNT
DEFINE(#[NUMCOUNT#], INCR(NUMCOUNT))
```

The expanded text is always rescanned, because of the possibility that it contains more macros in need of processing. The current example is just such a case. Let's take a look at the interesting intermediate states which the second scanning pass goes through:

```
0
DEFINE(#[NUMCOUNT#], INCR(NUMCOUNT))
```

Here, the first line has been rescanned, replacing NUMCOUNT with its previously defined value of 0.

```
0
DEFINE(NUMCOUNT, INCR(NUMCOUNT))
```

At this point, we have stripped off the protection brackets from the first parameter. The second parameter has not yet been scanned.

```
0
DEFINE(NUMCOUNT, INCR(0))
```

The second parameter has been partially scanned, replacing NUMCOUNT with its current value, 0.

```
0
DEFINE(NUMCOUNT, 1)
```

The second parameter has now been completely scanned, changing INCR's 0 argument to a 1. The DEFINE can now be invoked, which resets NUMCOUNT to be equivalent to 1.

```
0
```

Here, the effect of NUMCOUNT's redefinition has been propagated. The DEFINE leaves behind its effect, but not a value. The only value left for NEXTNUM to return is the 0. The next time NEXTNUM is invoked, NUMCOUNT will already be defined to be 1 and the whole process can start over again. This is the sense in which NUMCOUNT has been used as a macro-time variable of NEXTNUM. Its value is saved and continually updated each time NEXTNUM is invoked.

Two more macro primitives require our attention:

IFELSE compares its first two parameters, and macro-evaluates to its third parameter if they are equal, or its fourth if they are not. The result of the IFELSE directive is then rescanned.

Example:

```
        DEFINE(FLAG,ON)

        IFELSE(FLAG,ON, #[INCLUDE(EXTRADEFS)#], )
```

In this example, the DEFINE of FLAG might exist at the beginning of a program, which depending upon some compile-time conditions may be set to ON or OFF. When the IFELSE is encountered, if FLAG is defined to be ON, the INCLUDE will be invoked. If FLAG is OFF (or anything else, for that matter) the IFELSE evaluates to the fourth parameter, which in this example is null.

SUBSTR takes three parameters: a string and two numbers. SUBSTR evaluates to the SUBSTRing of the first parameter which starts with the character position indicated by the second parameter (zero-relative) for as many characters as are indicated by the third parameter. If the third parameter is omitted, it is implicit that the substring should continue until the end of the first parameter.

Example:

```
        SUBSTR(Kiss me; I'm a DASL programmer, 9, 18)

expands to: I'm a DASL program


        SUBSTR(Give me liberty or give me a substring, 19)

expands to: give me a substring
```

PASCAL and some other languages have a feature known as "enumerated types." This basically means that I can define a type, for instance, COLOR, and define various unique instances of that type, for instance, RED, GREEN and BLUE. In PASCAL, one would simply say:

```
TYPE COLOR = (RED, GREEN, BLUE);
```

In DASL one could express this as:

```
        TYPDEF COLOR BYTE;

        DEFINE(RED, 0)
        DEFINE(GREEN, 1)
        DEFINE(BLUE, 2)
```

It would, however, be better if we had a shorthand notation for the creation and management of enumerated types. ENUM and ENUMV are macros which do just that:

```
        DEFINE(ENUMV, #[
           IFELSE(#2,,,#[DEFINE(#2, #1)
                          ENUMV(INCR(#1),#3,#4,#5,#6,#7,#8,#9)#])
           #])

        DEFINE(ENUM, #[ENUMV(0,#1,#2,#3,#4,#5,#6,#7,#8)#]BYTE)
```

Using these macros in our COLOR example, one would simply say:

```
        TYPDEF COLOR ENUM(RED, GREEN, BLUE);
```

The effect of this declaration would be equivalent to the full set of TYPDEF and DEFINEs shown above.

Rather than stepping through all of the details of ENUM's macro evaluation, let's just take a look at some of the highlights. ENUMV is a recursive macro. Keeping in mind the principles of recursive programming discussed in Chapter VI, take a good look at the IFELSE. The IFELSE is the "bottom" of the recursive process; the light at the end of the Klein Bottle. Each successive ENUMV which gets spawned by the recursion has its parameter list shifted down one position from its parent's corresponding list (parameter #3 is put in the second position, and so on). In this fashion, the parameter list is getting "chomped" shorter with each invocation. At some point, parameter

#2 will become null and ENUMV will stop giving birth to new ENUMV's. With each level of recursion, parameter #1 (the number which is being associated with each instance of the type) gets incremented.

ENUM is simply a neat package which seeds ENUMV with 0, hands off the actual names of the enumerated instances and leaves behind the literal string **BYTE**, for use by the declaration in which ENUM is invoked. For those of you who want to use ENUM, please note that it is already defined in D$INC (see Appendix 2).

For those of you would are interested in looking at more macro definitions of non-trivial complexity, the I/O package support file, SIOINC (see Appendix 0), is a good place to start.

One last thought about dealing with DASL macros. The compiler has a facility which will expand your macro-bearing "SUPERDASL" program into a fully macro-evaluated, plain vanilla DASL source file. To invoke this bit of magic, you must provide an "EXP" file specification on the DASL command line. For instance:

```
DASL BIGBANG,EXP=UNIVERSE
```

The program called BIGBANG will be compiled normally, but the full source code, including the expanded version of all of BIGBANG's macros, will be written into the file called UNIVERSE. You might try compiling D$INC or SIO with the expand option active and take a look at what pops out. Some day you'll be glad you did.

X. Principia Apologia

In the pages that follow you will find some amusing, and hopefully enlightening DASL programs. But before we get to them, there are a few fine points which should be clarified:

The tutorial took a lot of shortcuts, and left out a number of fascinating things in the interest of brevity. DASL has a rich set of operators, logical, bitwise, arithmetic--even conditional. You are urged to thumb through the DASL Language Reference Guide while reading over the programs in the next section.

Another approximation which should be resolved: often the word "variable" was used when the term "lvalue" was more correct. An lvalue is an expression which refers to an entity which lives in memory (and can therefore appear on the left side of an assignment statement, hence the name). Variables are just one sort of lvalue. A string constant is an lvalue, and the following statement is meaningful in DASL:

```
CHARPOINT := &('this is a string constant'[0]);
```

Other examples of lvalues would include expressions such as:

```
(P + 10)^
STRUCTP^.FIELD
ARRAY[I]
```

and anything else which represents a value in memory. The DASL Language Reference Guide explains the matter and uses the technical meaning of lvalue in a consistent manner. Just thought you'd like to see it here first.

We also breezed over global variables, probably because good modular programming practice tends to keep them to a minimum. But they are valuable when used properly. Global variables can be declared anywhere in a program, outside of the body of a function. They are necessarily STATIC, since they are not attached to anything which can get re-initialized. Global variables may be imported from other modules by prefacing their declaration with the keyword EXTERN or can be exported by prefacing their definitions with the keyword ENTRY.

Which brings up one last point: the difference between definitions and declarations. A **definition** causes some new memory entity to be created. A **declaration** causes some already existing entity to become visible in the declaring context. Most often, definitions and declarations take place at the same time. But when we consider import/export, it is an important distinction to bear in mind.

With these fine points taken care of, you should now be in a position to begin reading and writing DASL programs of some complexity. In the next section, we will take a look at two "big" DASL programs (big in this case means I couldn't fit all of the source code onto a single page!). Feel free to play with them, to improve upon them (there's certainly room), to break them and put them back together again. And I heartily recommend that you browse through the DASL Language Reference Guide (see Appendix 1) at random and frequent intervals while doing so. It is an excellent source of compact and precise information.

SAMPLE DASL PROGRAMS

1. DATES

The following program defines a new data type called DATE (see Chapter 4, **Anatomy of a DASL Program**, for the sections dealing with STRUCTURES and TYPDEF). In this program, the user is prompted to fill in the information corresponding to several DATE structures. The DATEs are then sorted in ascending order of year/month/day and redisplayed on the screen.

The actual sorting algorithm used is the same bubble sort that we saw in Chapter V. Once again, notice that MAIN is logically very sparse, farming out all of the real work. Also notice the use of nested IF/THEN/ELSE constructs in the sorting function called ASCENDING. Since we are accustomed to seeing semicolons nearly everywhere in DASL, it might be worthwhile to point out that at the end of the THEN portion of an IF/THEN/ELSE construct there is - (dramatic pause) - no semicolon. This is because the entire IF/THEN/ELSE construct is taken as a single statement, and the ELSE clause at that point has yet to be stated.

There is a simpler way to write the ASCENDING function. One could derive a unique value for each possible calendar date and simply compare them. You might like to rewrite the function in that manner as a coding exercise. Or you might not.

Toward the beginning of the program, you'll notice a declaration which begins with the keyword EXTERN. This means that the entity which follows (in this case a function named INITCHAR) is not defined in the current module. It is to be found in another module somewhere, and its /REL file must be linked with this module's /REL file to make the program work (see The Joy of Running). The compiler, however, does need to know the type of the entity, so that it can properly handle data type compatibility and expression evaluation. If you recall, the type of a function is defined by the types of the parameters it is called with and the type of the result which it returns. The syntax of the language also demands that in such a function declaration, the input parameters are given names (exactly like those of a normal function definition), even though those names will never be used.

INITCHAR simply initializes memory starting with the position pointed to by P, for as many characters as are specified by SIZE with the character indicated by FILLER. You should take a minute out to write INITCHAR and generate a /REL module for it. Then make it the first member of your own private utility library.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(SIOINC)

DEFINE(NDATES,5)
DEFINE(WIDTH,80)
DEFINE(EEOF,0200)
DEFINE(EEOL,0201)
DEFINE(HOME,#[C,$H, C,0, C,$V, C,0#])


EXTERN INITCHAR(P∧ CHAR, SIZE INT, FILLER CHAR);


TYPDEF DATE STRUCT {
              MONTH,DAY,YEAR INT;
              MESSAGE[WIDTH] CHAR;
              };



SWITCH(P1, P2 ∧ DATE) :=
VAR TEMP DATE;
{
   TEMP := P1∧;
   P1∧ := P2∧;
   P2∧ := TEMP;
   };



ASCENDING(P1, P2 ∧ DATE) BOOLEAN :=
{
   RESULT := TRUE;
   IF P1∧.YEAR > P2∧.YEAR THEN RESULT := FALSE
    ELSE {
      IF P1∧.YEAR = P2∧.YEAR THEN {
         IF P1∧.MONTH > P2∧.MONTH THEN RESULT := FALSE
          ELSE {
            IF P1∧.MONTH = P2∧.MONTH THEN {
               IF P1∧.DAY > P2∧.DAY THEN RESULT := FALSE;
               };
            };
         };
      };
   };
```

```
SORTDATES(DP, MAXDP ^ DATE) :=
VAR P ^ DATE;
{
   LOOP {
      P := DP;
      LOOP { WHILE (P + 1) <= MAXDP;
         IF ~ASCENDING(P, P + 1) THEN SWITCH(P, P + 1);
         P++;
         };
    WHILE --MAXDP > DP;
      };
   };

GETDATES(DP ^ DATE) :=
{
   INITCHAR(&(DP^.MESSAGE[0]), SIZEOF (DP^.MESSAGE), ' ');
   D$WRITE(SCREEN, HOME);
   D$WRITE(SCREEN, C,EEOF);
   D$WRITE(SCREEN, S,'MONTH? (1-12): ');
   DP^.MONTH := D$READI(KEYBD);
   D$WRITE(SCREEN, LN, S,'DAY? (1-31): ');
   DP^.DAY := D$READI(KEYBD);
   D$WRITE(SCREEN, LN, S,'YEAR? (00-99): ');
   DP^.YEAR := D$READI(KEYBD);
   D$READC(KEYBD);
   D$WRITE(SCREEN, LN, S,'ENTER MESSAGE FOR THIS DATE:', LN);
   D$READS(KEYBD, &(DP^.MESSAGE[0]), SIZEOF DP^.MESSAGE);
   };

SHOWDATES(DP ^ DATE) :=
VAR DPCOUNT INT;
    TIMER UNSIGNED;
{
   DPCOUNT := 0;
   D$WRITE(SCREEN, HOME);
   D$WRITE(SCREEN, C,EEOF);
   D$WRITE(SCREEN, S,'Displaying DATES in sorted order:', LN);
   LOOP { WHILE DPCOUNT < NDATES;
      D$WRITE(SCREEN, C,$H, C,0, C,$V, C,++DPCOUNT);
      D$WRITE(SCREEN, I,(DP^.MONTH), C,'/');
      D$WRITE(SCREEN, I,(DP^.DAY), C,'/');
      IF DP^.YEAR < 10 THEN D$WRITE(SCREEN, C,'0');
      D$WRITE(SCREEN, I,(DP^.YEAR), C,EEOL, LN);
      D$WRITE(SCREEN, S,(DP++^.MESSAGE), C,EEOL);
      TIMER := 64000;
      LOOP WHILE TIMER-- > 0;
      };
   };
```

```
ENTRY MAIN() :=
VAR DATEARRAY [NDATES] DATE;
    DCOUNT INT;
{
   DCOUNT := 0;
   LOOP { WHILE DCOUNT < NDATES;
      GETDATES(&DATEARRAY[DCOUNT++]);
      };
   SORTDATES(&DATEARRAY[0], &DATEARRAY[NDATES - 1]);
   SHOWDATES(&DATEARRAY[0]);
   };
```

2. The Game of LIFE

John Conway's game of LIFE has probably been implemented for every respectable computer (and some questionable ones) and in every conceivable programming language. In fact, our own Harry Pyle implemented a LIFE game on the 2200 back around 10 B.D. ("Before DASL").

The game of LIFE is very simple. Every character position on the screen represents a "cell". In any given generation a cell is either alive (and therefore visible) or dead. From generation to generation the following algorithm is used to decide which cells survive, which die, and which are born:

1. If a cell is currently alive and either 2 or 3 of its 8 neighbors (that is, the 8 other cells which touch it) are alive, the cell will survive into the next generation.

2. If a cell is currently dead, and it has exactly 3 live neighbors, it will be born in the next generation.

```
---------------------------------
|          |          |          |
| NEIGHBOR | NEIGHBOR | NEIGHBOR |
|    1     |    2     |    3     |
|----------|----------|----------|
|          |          |          |
| NEIGHBOR |   CELL   | NEIGHBOR |
|    4     |          |    5     |
|----------|----------|----------|
|          |          |          |
| NEIGHBOR | NEIGHBOR | NEIGHBOR |
|    6     |    7     |    8     |
---------------------------------
```

That's it. From this simple set of constraints and an initial "seeding" of the world with live cells, wonderful patterns emerge and evolve. Formations such as "traffic lights" and "gliders" are familiar and well loved denizens of the LIFE world. Sections of the world can become totally static, only to become reactivated when other formations absorb them. At any rate, the game is fascinating and sometimes profound.

```
INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(SIOINC)


DEFINE(EEOF,0200)
DEFINE(WIDTH,80)
DEFINE(HEIGHT,12)
DEFINE(DEAD,' ')
DEFINE(LIVE,'*')


TYPDEF WORLD [HEIGHT][WIDTH] BOOLEAN;


EXTERN INITCHAR(P ^ CHAR, SIZE INT, FILLER CHAR);


NEXTGEN(THIS, NEXT ^ WORLD) :=
VAR I, J, X, Y, SUM INT;
{
   I := 0;
   LOOP { WHILE I < HEIGHT;
      J := 0;
      LOOP { WHILE J < WIDTH;
         SUM := 0;
         X := I - 1;
         LOOP { WHILE X <= I + 1;
            Y := J - 1;
            LOOP { WHILE Y <= J + 1;
               IF X >= 0 & X < HEIGHT & Y >= 0 & Y < WIDTH THEN
                              SUM += THIS ^ [X][Y];
               Y++;
               };
            X++;
            };
         IF THIS^[I][J] THEN NEXT^[I][J] := ((SUM = 3) | (SUM = 4))
          ELSE NEXT^[I][J] := (SUM = 3);
         J++;
         };
      I++;
      };
   };
```

```
BINARIZE(P ∧ CHAR) :=
VAR COUNT INT;
{
   COUNT := 0;
   LOOP { WHILE COUNT < WIDTH;
      P∧ := ~(P∧ = DEAD);
      COUNT++;
      P++;
      };
   };



INITWORLD(P ∧ WORLD) :=
VAR VCOUNT INT;
    INPCOUNT INT;
{
   VCOUNT := 0;
   D$WRITE(SCREEN, C,$H, C,0, C,$V, C,0);
   D$WRITE(SCREEN, C,EEOF);
   LOOP { WHILE VCOUNT < HEIGHT;
      INITCHAR(&(P∧[VCOUNT][0]), WIDTH, DEAD);
      D$WRITE(SCREEN, C,$H, C,0, C,$V, C,VCOUNT);
      D$WRITE(SCREEN, F);
      INPCOUNT := D$READS(KEYBD, &(P∧[VCOUNT][0]), WIDTH);
      IF --INPCOUNT < WIDTH THEN P∧[VCOUNT][INPCOUNT] := DEAD;
      BINARIZE(&(P∧[VCOUNT][0]));
      VCOUNT++;
      };
   };



CONVERT(BP ∧ BOOLEAN, CP ∧ CHAR) :=
VAR COUNT INT;
{
   INITCHAR(CP, WIDTH, DEAD);
   COUNT := 0;
   LOOP { WHILE COUNT++ < WIDTH;
      CP++∧ := (BP++∧ = TRUE) ? (LIVE) : (DEAD);
      };
   };
```

```
SHOWGEN(P ∧ WORLD) :=
VAR LIFELINE [WIDTH] CHAR;
    VCOUNT INT;
{
   VCOUNT := 0;
   LOOP { WHILE VCOUNT < HEIGHT;
      CONVERT(&P∧[VCOUNT][0], &LIFELINE[0]);
      D$WRITE(SCREEN, C,$H, C,0, C,$V, C,VCOUNT);
      D$WRITE(SCREEN, S,LIFELINE, F);
      VCOUNT++;
      };
   };



ENTRY MAIN() :=
VAR THIS, NEXT, TEMP ∧ WORLD;
    GEN0, GEN1 WORLD;
{
   THIS := &GEN0;
   NEXT := &GEN1;
   INITWORLD(THIS);
   LOOP {
      NEXTGEN(THIS, NEXT);
      SHOWGEN(THIS);
      TEMP := THIS;
      THIS := NEXT;
      NEXT := TEMP;
      };
   };
```

This implementation lets the user enter an initial pattern of live cells, and then runs an infinite loop of generations. The initial entry facility is rather crude: the user types spaces for dead cells and any non-space character to signify a live cell. No random cursor positioning is provided. Hitting the ENTER key will terminate a particular horizontal line and will fill all trailing positions with dead cells. If you are feeling ambitious, I strongly suggest that you build a more sophisticated pattern editor for the LIFE game, and allow the loop to be exited temporarily, so the user can manually reshape the current generation.

The program carries a representation of each generation as a two dimensional array of BOOLEAN's, that is, a 1 (or TRUE) for a live cell and a 0 (or FALSE) for a dead cell. This makes some aspects of the programming simpler, but means that a conversion process must be invoked before the generation is displayed on the screen. This conversion is done with the ternary or conditional operator (see the CONVERT function). The expression containing the conditional operator is made up of three smaller expressions. If the first expression is evaluated to be logically TRUE (that is, non-zero in value) then the total expression takes the value of the component expression following the question mark. If the first expression is logically FALSE (that is, it is equal to zero), the total expression takes the value of the component expression following the colon. You can think of the conditional operator as an expression level flow of control construct, similar in operation to the statement level IF/THEN/ELSE construct.

Another interesting feature of this program is that it uses logical values directly. For instance, look at this line, taken from the NEXTGEN function:

```
IF THIS^[I][J] THEN NEXT^[I][J] := ((SUM = 3) | (SUM = 4))
```

Since the array pointed to by THIS contains only BOOLEANs, the value of THIS^[I][J] must be either a 1 or a 0. Therefore, it is sensible to use its value directly to motivate the THEN/ELSE decision. There is no need to compare the value with TRUE or FALSE to force a logical interpre·ation. Also, we assign into NEXT^[I][J] the result of the expression ((SUM = 3) | (SUM = 4)). That is, if SUM = 3 is TRUE or SUM = 4 is TRUE, then NEXT^[I][J] will get assigned a TRUE, namely, a 1. If the logical expression comes up FALSE, then NEXT^[I][J] will have a 0 assigned into it. In both of these cases, we are using the fact that these expressions have a direct logical interpretation. This is why the program operates on BOOLEANs and does a post-compute conversion to displayable characters.

Although DASL produces rather good object code, some source level optimizations are possible. Some of these optimizations, however, come at the expense of the readability and maintainability of the code. On the following pages we will see two optimized implementations of the NEXTGEN function. Because the interface between NEXTGEN and the rest of the LIFE program is so well defined, you can simply swap any of these implementations and recompile.

The first optimization is rather reasonable, and is concerned with tightening of the critical loop. The second optimization attempts to re-write the NEXTGEN function purely in terms of pointers. This strategy turns out to be of dubious merit, for while it improves the speed of execution by some 10%, it also renders the source code nearly unreadable. This is just another one of those situations where common sense (and a healthy respect for the role of software maintenance) will be your best guide.

```
/* This version of NEXTGEN is almost identical to the previous
one.  The only differences are that I, J, X, Y and SUM have been
defined as BYTEs (8 bit unsigned values) and one of the loops has
been significantly tightened.  Since we know that the values won't
exceed 255, we can save a little space using 8 bit BYTEs rather
than 16 bit INTs. */


NEXTGEN(THIS, NEXT ^ WORLD) :=
VAR I, J, X, Y, SUM BYTE;
{
   I := 0;
   LOOP { WHILE I < HEIGHT;
      J := 0;
      LOOP { WHILE J < WIDTH;
         SUM := 0;
         X := I - 1;
         LOOP { WHILE X <= I + 1;
            IF X >= 0 & X < HEIGHT THEN {
               Y := J - 1;
               LOOP { WHILE Y <= J + 1;
                  IF Y >= 0 & Y < WIDTH THEN
                     SUM += THIS ^ [X][Y];
                  Y++;
                  };
               };
            X++;
            };
         IF THIS^[I][J] THEN NEXT^[I][J] := ((SUM = 3) | (SUM = 4))
           ELSE NEXT^[I][J] := (SUM = 3);
         J++;
         };
      I++;
      };
   };
```

```
/* This version of NEXTGEN is rather different from the previous
two.  It deals with the evaluations totally in terms of pointers.
In general, the compiler generates shorter, and consequently
faster code if pointers are used.  But the pointer representation
of a two dimensional data element can get rather hairy, as you
should be able to see from the code. */


NEXTGEN(THIS, NEXT ^ WORLD) :=
VAR I, J, X, Y, T, W, NEXTP ^ CHAR;
    SUM BYTE;
{
   I := &THIS^[0][0];
   NEXTP := &NEXT^[0][0];
   LOOP { WHILE I < &THIS^[HEIGHT][0];
      J := I;
      LOOP { WHILE J < I + WIDTH;
         SUM := 0;
         X := I - WIDTH;
         LOOP { WHILE X <= I + WIDTH;
            IF X >= &THIS^[0][0] & X <= &THIS^[HEIGHT-1][WIDTH-1]
             THEN {
               Y := X + (J - I) - 1;
               T := Y + 3;
               W := X + WIDTH;
               LOOP { WHILE Y < T;
                  IF (Y >= X) & (Y < W) THEN SUM += Y^;
                  Y++;
                  };
               };
            X += WIDTH;
            };
         IF J^ THEN NEXTP^ := ((SUM = 3) | (SUM = 4))
          ELSE NEXTP^ := (SUM = 3);
         NEXTP++;
         J++;
         };
      I += WIDTH;
      };
   };
```

TECHNICAL APPENDICES

Appendix 0. SIO

SIO is a simple screen and keyboard only I/O package developed by Gene Hughes specifically for this tutorial. In the pages that follow, the complete source code for SIO and its associated INCLUDE file, SIOINC, will be shown. Before looking at the code, let's lay out an operational description of each of the available functions:

D$READC(KEYBD) - Returns a single character from the keyboard.

D$READI(KEYBD) - Returns an integer from the keyboard. Accepts an ASCII string and converts it to an INT.

D$READS(KEYBD, POINTER, LENGTH) - Reads a string from the keyboard into memory, starting at the character position pointed to by POINTER, for as many characters as are specified by LENGTH. A logical end of record character will be embedded in the array if the number of characters entered is less than LENGTH. D$READS also returns the number of characters keyed in, including the logical end of record if it is embedded.

D$READCs and D$READIs (unless followed by other D$READIs) will leave an end of record character sitting in the buffer which you must bump past before you can successfully perform another I/O call. Eating it up with an extra D$READC is usually sufficient.

D$WRITE(SCREEN, parameters) - Outputs the indicated information to the screen. D$WRITE works in terms of parameter groups:

To write a character:	**D$WRITE(SCREEN, C,CHARACTER)**	
To write an integer:	**D$WRITE(SCREEN, I,INTEGER)**	
To write a string:	**D$WRITE(SCREEN, S,STRING)**	or
	D$WRITE(SCREEN, S,'Literal string')	
To write a string of specific length:	**D$WRITE(SCREEN, SL,POINTER,LENGTH)**	
To write a new line:	**D$WRITE(SCREEN, LN)**	
To flush the buffer:	**D$WRITE(SCREEN, F)**	

The buffer will automatically be flushed by a new line or by a D$READ type instruction. Parameter groups may be used in combination inside of a single D$WRITE, as long as the total number of parameters does not exceed nine. For example:

```
D$WRITE(SCREEN, LN, SL,&TEXT[0],WIDTH, I,TOTAL)
```

SIOINC/TEXT - SIOINC is an INCLUDE file which contains the definitions required by the SIO package. Feel free to build upon SIO and SIOINC to develop new functions which you think are useful.

```
/* KEYBOARD AND DISPLAY SEQUENTIAL I/O DEFINITIONS */

TYPDEF D$FILET STRUCT {
   P ^ CHAR;
   C UNSIGNED;
   BUF [81] CHAR;
   };


EXTERN D$IN D$FILET;
DEFINE(KEYBD, &D$IN)

EXTERN D$READC (F ^ D$FILET) CHAR;

EXTERN D$READS (F ^ D$FILET, S ^ CHAR, N UNSIGNED) UNSIGNED;

EXTERN D$READI (F ^ D$FILET) INT;


EXTERN D$OUT D$FILET;
DEFINE(SCREEN, &D$OUT)

EXTERN D$WRITEC (F ^ D$FILET, C INT);

EXTERN D$WRITES (F ^ D$FILET, S ^ CHAR, N UNSIGNED);

EXTERN D$WRITEI (F ^ D$FILET, N INT);

EXTERN D$WRITEF (F ^ D$FILET);

DEFINE(D$T,#[IFELSE(SUBSTR(#1,0,1), ,#[D$T(SUBSTR(#1,1))#],
 #[IFELSE(SUBSTR(#1,0,1),
,#[D$T(SUBSTR(#1,1))#],#[D$W#1#])#])#])

DEFINE(D$WRITE,#[{
 D$T(#2)(#1,#3,#4,#5,#6,#7,#8,#9)}#])

DEFINE(D$WC,#[D$WRITEC(#1,#2);
 D$T(#3)(#1,#4,#5,#6,#7,#8)#])

DEFINE(D$WS,#[D$WRITES(#1,&(#2)[0],
 SIZEOF(#2));
 D$T(#3)(#1,#4,#5,#6,#7,#8)#])

DEFINE(D$WSL,#[D$WRITES(#1,#2,#3);
 D$T(#4)(#1,#5,#6,#7,#8)#])
```

```
DEFINE(D$WI,#[D$WRITEI(#1,#2);
 D$T(#3)(#1,#4,#5,#6,#7,#8)#])

DEFINE(D$WLN,#[D$WRITEC(#1,$LEOR);
 D$T(#2)(#1,#3,#4,#5,#6,#7,#8)#])

DEFINE(D$WF,#[D$WRITEF(#1);
 D$T(#2)(#1,#3,#4,#5,#6,#7,#8)#])

DEFINE(D$W, )
```

SIO/TEXT - This is the actual source code for the Sample I/O package used in this tutorial. Note that SIO is a standard DASL program, no different in principle from any of the others we have looked at so far.

```
/* KEYBOARD AND DISPLAY SEQUENTIAL I/O ROUTINES */

INCLUDE(D$INC)
INCLUDE(D$RMS)
INCLUDE(D$RMSWS)
INCLUDE(D$UFRWS)
INCLUDE(SIOINC)

HOR BYTE := 0;
VER BYTE := 11;


ENTRY D$IN :=  { NIL, 0 };

ENTRY D$READC :=
VAR   LEN BYTE;
{
   IF D$IN.C-- = 0 THEN {
      D$WRITEF(&D$OUT);
      LEN := SIZEOF D$IN.BUF;
      IF $GETLINE(&D$IN.BUF[0], 0, &LEN, &HOR, &VER) && D$CFLAG THEN
         $ERMSG();
      D$WRITEC(&D$OUT, $LEOR);
      LEN := SIZEOF D$IN.BUF - LEN;
      D$IN.BUF[LEN-1] := $LEOR;
      D$IN.P := &D$IN.BUF[0];
      D$IN.C := LEN - 1;
      };
   RESULT := D$IN.P++^;
   };

ENTRY D$READS :=
{
   RESULT := 0;
   LOOP {
   WHILE N--;
      S^ := D$READC(F);
      RESULT++;
   WHILE S++^ ~= $LEOR;
      };
   }.
```

```
ENTRY D$READI :=
VAR   SIGN INT;
      C CHAR;
{
   LOOP {
      C := D$READC(F);
   WHILE C = ' ' | C = $LEOR;
      };
   SIGN := 1;
   IF C = '-' THEN {
      SIGN := - 1;
      C := D$READC(F);
      };
   RESULT := 0;
   LOOP {
   WHILE '0' <= C & C <= '9';
      RESULT := C - '0' + RESULT * 10;
      C := D$READC(F);
      };
   RESULT *= SIGN;
   --F∧.P;
   F∧.C++;
   };


ENTRY D$OUT := { &D$OUT.BUF[0] };

ENTRY D$WRITEC :=
{
   IF C = $LEOR THEN {
      D$OUT.P++∧ := $EL;
      D$WRITEF(&D$OUT);
      }
   ELSE {
      IF D$OUT.P + (0200 <= C & C <= 0277)
       > &D$OUT.BUF[(SIZEOF D$OUT.BUF)-3] THEN D$WRITEF(&D$OUT);
      D$OUT.P++∧ := C;
      };
   };

ENTRY D$WRITES :=
{
   LOOP {
   WHILE N--;
      D$WRITEC(F, S++∧);
      };
   };
```

```
ENTRY RECURSIVE D$WRITEI :=
{
   IF N < 0 THEN {
      D$WRITEC(F, '-');
      N := - N;
      };
   IF N / 10 ~= 0 THEN D$WRITEI(F, (<UNSIGNED>N)/10);
   D$WRITEC(F, (<UNSIGNED>N)%10+'0');
   };

ENTRY D$WRITEF :=
{
   D$OUT.P^ := $ES;
   IF $PUTNOTX(&D$OUT.BUF[0], &HOR, &VER) && D$CFLAG THEN $ERMSG();
   D$OUT.P := &D$OUT.BUF[0];
   };
```

Appendix 1. The DASL Language Reference Guide

1. Introduction

DASL, Datapoint's Advanced Systems Language, is a high level systems programming language designed to replace assembly language for most programming tasks. The primary design goal was to provide a simple language with sufficient power for systems programming. A simple language results in easy understanding, fast compilation, and efficient code generation.

DASL takes most of its ideas from the languages C and PASCAL. The language has variable size scalar data, as well as pointer, array, structure, and function types. There are many expression operators for efficiently manipulating data. There are a few simple but powerful control constructs. Finally, the language includes a simple macro facility which can be used for many different functions ranging from constant definition to language extensions.

2. Lexical rules

The DASL tokens are identifiers, numbers, strings, and other symbols. Comments, blanks, and line ends may be used freely between tokens.

Any line starting with a period, star, or plus character is considered a comment line and is completely ignored by the compiler, with the exception of a line beginning with */ (see comments, section #2.5).

2.1 Identifiers

An identifier is a sequence of characters chosen from upper or lower case letters, digits, and the characters $ and _ (underscore); the first character may not be a digit. The first 29 characters and last character are significant for determining uniqueness. Only the first seven characters and last character are significant for identifiers passed to the LINK utility as ENTRY or EXTERN names.

2.2 Reserved and predefined identifiers

The following identifiers are reserved for use as keywords:

CASE	IF	SYSTEM
DEFAULT	LOOP	THEN
ELSE	RECURSIVE	TYPDEF
ENTRY	SIZEOF	UNION
EXTERN	STATIC	VAR
FAST	STRUCT	WHILE
GOTO		

The following identifiers are predefined by the compiler but may be redefined:

BOOLEAN	IFELSE	LONG
BYTE	INCLUDE	SUBSTR
CHAR	INCR	UNSIGNED
DEFINE	INT	

The identifier RESULT is predefined inside functions but may be redefined.

2.3 Numbers

A decimal number is a sequence of digits not beginning with the digit 0. An octal number is a sequence of the digits 0 through 7 beginning with the digit 0. A hexadecimal number is a sequence of digits and the characters **A** through **F** (upper or lower case) beginning with the characters 0x or 0X.

2.4 Strings

A string is a sequence of characters surrounded by single quotes, such as 'abc'. Within a string a quote is represented by two consecutive quotes. An end of line in the middle of a string is ignored.

2.5 Comments

A comment begins with the characters /* and ends with the characters */. Comments may be nested.

3. Syntax notation

```
identifier-list -> identifier
                -> identifier , identifier-list
```

In this example, an identifier list is either an identifier, or an identifier followed by a comma followed by an identifier list. Words in upper case are keywords in the language. The subscript $_{opt}$ on an item indicates that the item is optional.

4. Types

A type is used in a variable declaration (#7.2) to determine the amount of storage allocated as well as the interpretation of the variable when it is used in an expression. A type is also used in a function declaration (#7.3) to specify the parameters to and result of the function. The TYPDEF declaration (#7.1) is used to give a type a name which may be then used as a type.

Every expression has an associated type which helps determine the meaning of the expression and is used to detect errors. The type of an expression may be changed by the use of a type in a cast (#5.18).

4.1 Named types

```
type -> identifier
```

An identifier is a type, provided it has been predefined by the compiler or has been defined by a TYPDEF declaration (#7.1).

4.2 Scalar types

A scalar type describes a value which is an integral number. Associated with each scalar is its length in bytes and whether it is signed (can be negative). The following scalar types are predefined by the compiler:

BOOLEAN	1 byte	unsigned
CHAR	1 byte	unsigned
BYTE	1 byte	unsigned
UNSIGNED	2 bytes	unsigned
INT	2 bytes	signed
LONG	4 bytes	signed

The only scalar types which may be defined by a program are types equivalent to the above, defined by a TYPDEF declaration.

4.3 Pointer types

```
type -> ∧ type
```

A pointer type describes a value which is either an address of an object in memory or the value zero, representing a null pointer. The type of the object to which the pointer points is given by type following the ∧ symbol; it may be any type. If the type following the ∧ symbol is an identifier, it may be a forward reference which is defined later by a TYPDEF declaration; the identifier may even never be defined, unless its definition is needed by the compiler to determine the meaning of an expression involving the pointer type. The size of a pointer is two bytes for the 5500 instruction set.

```
Examples:  ∧ NODE
           ∧ ∧ CHAR
```

4.4 Array types

type -> [constant-expression$_{opt}$] type

An array is an aggregate type containing a number of components of the type given following the] symbol. The number of components is given by the expression, which must be a compile time constant. The components are numbered from zero to one less than the number of components specified as the upper bound. The bound may be omitted in certain variable declarations (#7.2). If the component type is also an array type the result is a multidimensional array.

```
Examples:  [100] INT
           [32] [8] ∧ CHAR
```

4.5 Structure types

```
type -> structure
```

structure -> STRUCT { struct-decl-list$_{opt}$ }
          -> UNION { struct-decl-list$_{opt}$ }

```
struct-decl-list -> struct-declaration ;
                 -> struct-declaration ; struct-decl-list

struct-declaration -> identifier-list type
                   -> structure
```

A STRUCT is an aggregate consisting of a sequence of named members. A UNION is an object which may, at a given time, contain any one of several members. STRUCT and UNION types have the same syntax.

The struct-decl-list is a sequence of declarations for the members of the structure. The identifiers name the members of the structure. Within a STRUCT, successive members have successive positions; members have increasing offsets. In a UNION, each member starts at the beginning of the object at offset zero, and the UNION is large enough to hold the largest member.

One of the declarations making up a structure type may be an unnamed STRUCT or UNION; in that case the members of the unnamed structure may be referred to as part of the outer structure.

```
Example:   TYPDEF NODE STRUCT {
                   NAME [12] CHAR;
                   LEFT, RIGHT ^ NODE;
                   TYP INT;
                   UNION {
                      STR [8] CHAR;
                      S STRUCT { VAL1, VAL2 INT; };
                      };
                   };
           N NODE;
```

This example defines a type containing an array of 12 characters, two pointers to the same type, an integer, and an array which overlaps with two integers. After the variable N has been declared to be of type NODE, the expression N.NAME refers to the NAME field of the variable N; N.STR and N.S are also a valid references, but the VAL1 field must be accessed by N.S.VAL1 because the structure containing VAL1 was named.

4.6 Function types

type -> (param-decl-list$_{opt}$) type$_{opt}$

```
param-decl-list -> param-declaration
                -> param-declaration , param-decl-list
```

param-declaration -> identifier-list type

A function type describes the parameters and results of a function call. The parentheses are required even if there are no parameters. The types in the parameter declaration list describe the types of arguments which are expected when a function of the specified type is called; these types are used by the compiler to check for a valid call.

The type following the) symbol specifies the type returned by the function type; if this type is omitted, the function returns no result.

The parameter and result types must be scalar or pointer, but pointers to any types may be passed.

There are several restrictions on the use of a function type. The members of arrays and structures may not be functions, and a function parameter or result may not be a function type. However, a pointer to a function is valid in any of these places.

Example:
```
F (X ^ INT, Y, Z CHAR) INT :=
{  RESULT := X^ := X^ + Y + Z;
   };
```

In this example of a function declaration, the identifier F is followed by a function type which specifies three parameters and a result type. The part from the := on is the function body (#7.3) which computes the result returned by the function.

4.7 Type compatibility

Several operators require their operands to be of compatible type. Two types are said to be compatible if one of the following statements is true.

1. The types are scalar types.

2. The types are pointer types and the two types pointed to are compatible types of the same size.

3. The type of the constant number zero is compatible with any pointer type. In this context the number zero is considered to be a null pointer of the same type. The type of the constant number equal to the largest possible value of a pointer is also compatible with any pointer type.

3. The types are array types with the same upper bound and the component types are compatible types of the same size.

4. The types are equivalent types. Two types are equivalent if they were the same type declaration or one was a TYPDEF name for a type which is equivalent to the other.

Example:

```
TYPDEF S STRUCT { A, B INT; };
TYPDEF F ( X INT );

I INT;  I2 INT;
PI ^ INT;  PI2 ^ INT;  PF ^ F;  PC ^ CHAR;
AC [2] CHAR;  AAC [2] [1] CHAR;  AI [2] INT;
S1 S;  S2 S;  S3, S4 STRUCT { A, B INT; };
EXTERN F1 F;

F2 ( X INT ) :=
{  I  := I2;
   PI := PI2;
   PI := PF;    /* error, don't point to compatible types */
   PI := PC;    /* error, don't point to same size        */
   PI := 0;
   AC := '12';
   AC := '1';   /* error, array bounds differ             */
   AC := AAC;   /* error, component types not compatible  */
   AC := AI;    /* error, component types different size  */
   S1 := S2;
   S2 := S3;    /* error, not equivalent types            */
   S3 := S4;
   PF := & F1;
   PF := & F2;  /* error, don't point to equivalent types */
   };
```

5. Expressions

Expressions are used to compute values to be used in statements or declarations, or to perform actions such as function calls. Every expression has an associated type which helps determine the meaning of the expression and is used for error checking.

Except as noted in the description of particular operators, the order of evaluation of operands within expressions is undefined.

5.1 Lvalues and variables

An lvalue is an expression referring to an object in memory. The term "lvalue" comes from the assignment expression, in which the left operand must be an lvalue. A variable is an object which may be modified. An identifier is an lvalue referring to a variable. A string constant is an lvalue, but it does not refer to a variable. An integer constant is not an lvalue or variable.

The discussion of each operator below indicates if the operator expects lvalue or variable operands, and if the result is an lvalue or variable.

5.2 Arithmetic conversions

Many arithmetic operators perform type conversions on their scalar operands. These conversions are described here.

First any operands of type BOOLEAN, CHAR, or BYTE are converted to UNSIGNED by zero fill.

Then if either operand is type LONG, that is the type of the result. If the other operand is not type LONG, it is converted to LONG, by sign extension if type INT, otherwise zero fill.

Otherwise if either operand is type INT, both operands are considered to be type INT, and that is the type of the result. Care is required when INT and UNSIGNED operands are mixed.

Otherwise both operands are UNSIGNED, and that is the type of the result.

5.3 Operator precedence

The following table lists the operator precedence and the implied grouping when several operators of the same precedence appear at the same level. The operators listed first have the highest precedence and are performed first; operators listed on the same line have the same precedence. Parentheses may be used to modify precedence or grouping.

post-unary:	^ [] . () ++ --	left to right
pre-unary:	++ -- & - ~~ ~ SIZEOF <type>	right to left
binary:	* / %	left to right
	+ -	left to right
	<< >>	left to right
	&&	left to right
	|| !!	left to right
	= ~= < > <= >=	left to right
	&	left to right
	|	left to right
ternary:	?:	right to left
binary:	:= op=	right to left
	,	left to right

5.4 Constant expressions

Array bounds and case labels are required to be compile time constant expressions. These are expressions which only involve scalar constants and the arithmetic operators in sections #5.15 through #5.25.

5.5 Number

```
expression -> number
```

An integral number is a constant expression. Its type is normally UNSIGNED if the value will fit in sixteen bits, otherwise LONG. The constant values 0 and the largest possible pointer value may also be considered as pointers in the context described in section #4.7.

5.6 String

```
expression -> string
```

A string constant is an expression. A string normally has the type array of CHAR, but a string consisting of a single character may also be considered to be of UNSIGNED type. The string is initialized with the ASCII value of the characters in the string. A string is an lvalue.

5.7 Identifier

```
expression -> identifier
```

An identifier is an expression, provided it has been declared in a variable or function declaration. Its type is specified in its declaration, and an identifier is an lvalue.

5.8 Parentheses and comma operator

```
expression -> ( expression-list )

expression-list -> expression
                -> expression , expression-list
```

Parentheses may be used around a single expression to override the precedence of operators; they do not otherwise change the meaning of an expression.

If the expression list contains more than one expression separated by commas, each comma operator causes the preceding expression to be evaluated, but the value is discarded. The last operator performed by the expression must be one that has a side effect, namely function call, ++, --, assignment, or comma. The result of a comma expression is the same as the right operand. The comma expression is useful in macros (#8.) which return expressions.

```
Examples:  (I+1)*2
           (I+=2,A[I])
```

5.9 Indirection operator

expression -> expression ^

The ^ operator means indirection; the operand must be a pointer, and the result is an lvalue referring to the variable to which the pointer points. If the type of the operand is pointer to T, then the type of the result is T.

```
Example:   VAR I INT;
               PI ^ INT;
           {   PI := & I;
               I := 1;
               PI^ := 1;
               };
```

In this example the first assignment causes PI to point to I; the last two assignments both have the same effect of assigning to I.

5.10 Subscripting operator

expression -> expression [expression]

The subscripting operator [] selects a member of an array. The left operand must be an lvalue referring to an array, and the right operand must be a scalar. The result is an lvalue referring to the selected member of the array. The first member of an array is numbered zero. If the type of the left operand is array of T, then the type of the result is T.

```
Example:   VAR AAC [32] [8] CHAR;
           {   AAC[0][5] := AAC[1][6];
               AAC[1] := '12345678';
               };
```

5.11 Field operator

expression -> expression . identifier

The operator . selects a member of a structure or union. The left operand must be an lvalue referring to a STRUCT or UNION, and the . is followed by an identifier naming a member of that STRUCT or UNION. The result is an lvalue referring to the named member of the structure or union.

```
Example:    VAR S STRUCT { A, B INT; };
            {   S.A := 0;
                };
```

5.12 Function call operator

expression -> expression (expression-list$_{opt}$)

```
expression-list -> expression
                -> expression , expression-list
```

The first operand of a function call must be an lvalue referring to a function. The remaining expressions, if any, are the actual parameters to the function. They must agree in number with the formal parameters specified in the function type, and each operand must be of compatible type with the corresponding formal parameter. The parentheses are required even if there are no parameters. Parameters are passed by value, but it is possible to pass a pointer to an object which is to be modified. The order of evaluation of the function arguments is undefined, and may vary with the particular code generator used. The type of the result is that specified in the function type, or none if the function type specifies no result. Recursive calls are permitted if the function being called has been declared RECURSIVE (#7.3).

```
Examples:   EXTERN F (X INT);
            VAR PF ^ (X CHAR);
            {   F(1);
                PF^('a');
                };
```

In this example PF is a pointer to a function; the expression PF^ yields an lvalue referring to a function, which is then called with one parameter.

5.13 Increment and decrement operators

```
expression -> expression ++
           -> expression --
           -> ++ expression
           -> -- expression
```

The ++ operator increments the variable referred to by its operand; the -- operator decrements its operand. The operand must be an lvalue, and may be a scalar or pointer. If it is a scalar, it is incremented or decremented by one. If it is a pointer, the address to which the pointer points is incremented or decremented by the length of the object to which the pointer points. The value of the expression is the value of the operand after the increment or decrement if the ++ or -- operator precedes the operand, or the value before the increment or decrement if the operator follows the operand.

```
Example:    VAR I, J INT;
                PC ^ CHAR;
            {   I++;
                J := --I;
                PC++^ := ' ';
                (++PC)^ := ' ';
                };
```

The parentheses are used in the last assignment because unary operators which follow an expression have a higher precedence than unary operators preceding an expression.

5.14 Address operator

```
expression -> & expression
```

The unary & operator yields a pointer to the object referred to by its operand, which must be an lvalue. If the type of the operand is T, the type of the result is pointer to T.

```
Example:    VAR I INT;
                PI ^ INT;
            {   PI := & I;
                I := 0;
                PI^ := 0;
                };
```

Once PI is set to point to I by the first assignment, the last two assignments both have the same effect of assigning to I.

5.15 Unary arithmetic operators

```
expression -> - expression
           -> ~~ expression
```

The - operator gives the negative of its operand. The result is type LONG if the operand is LONG, otherwise INT.

The ~~ operator gives the one's complement of its operand. The type of the result is LONG if the operand is type LONG, INT if the operand is INT, and otherwise UNSIGNED.

5.16 Unary not operator

```
expression -> ~ expression
```

The result of the operator ~ is 0 if its operand is nonzero, otherwise 1. The operand must be a scalar or pointer. The type of the result is UNSIGNED.

5.17 SIZEOF operator

```
expression -> SIZEOF expression
           -> SIZEOF < type >
```

The SIZEOF operator gives the size, in characters or bytes, of its operand, which may be an expression or a type surrounded by corner brackets. The result is an UNSIGNED constant. The ambiguous case typified by SIZEOF<type>-1 is interpreted as (SIZEOF<type>)-1 by requiring that when SIZEOF is followed by an expression, the expression must not begin with <type> (#5.18).

5.18 Cast operator

```
expression -> < type > expression
```

An expression preceded by a type in corner brackets causes the operand to be considered to be the specified type. If the expression is a scalar or pointer, and the cast type is also scalar or pointer, then the value of the expression is converted, and the result is not an lvalue if the size changes. Otherwise the expression must be an lvalue, and the result is an lvalue referring to an object of the cast type at the memory address referenced by the expression. In this case, if the operand is a string constant, it will be padded with blanks if cast to a longer type; such a cast to a shorter type is an error.

Example: `<^CHAR>P := ALLOC(SIZEOF P^);`

In this example P is a pointer and ALLOC is a storage allocator which returns a pointer to a CHAR. The ALLOC call obtains enough storage for whatever P points to; the pointer is assigned to P, which has been cast to the proper type.

5.19 Multiplicative operators

```
expression -> expression * expression
           -> expression / expression
           -> expression % expression
```

The * operator indicates multiplication. The / operator indicates division. The % operator yields the remainder after the division of the first operand by the second. Both operands must be scalar. The arithmetic conversions described in #5.2 occur. For / and %, the operation is signed if either operand is signed, otherwise unsigned. The value of / and % is undefined if the second operand is 0, but no error occurs. The remainder has the same sign as the dividend. It is always true that A/B*B + A%B equals A, if B is not 0. The multiplicative operators group left to right.

5.20 Additive operators

```
expression -> expression + expression
           -> expression - expression
```

The + operator yields the sum of its operands. The - operator gives the difference of its operands. Both operators group left to right.

For +, the operands may be scalars, in which case the arithmetic conversions described in #5.2 occur, or they may be a pointer and a scalar. In the second case the pointer is incremented by the value of the scalar times the size of the object to which the pointer points, and the result is a pointer of the same type.

For -, if the operands are scalars, the result is type LONG if either operand is LONG, otherwise INT. If a scalar is subtracted from a pointer, the result is a pointer of the same type, given by the value of the pointer minus the value of the scalar times the size of the object to which the pointer points. If two pointers of compatible type are subtracted, the result is a signed scalar given by the difference of the two pointer values, divided by the size of the object to which they point.

```
Example:    VAR I INT;
                P, Q ^ INT;
                A [10] INT;
            {   P := & A[2];
                Q := & A[4];
                (P+1)^ := 0; /* assigns to A[3] */
                (P-2)^ := 0; /* assigns to A[0] */
                I := Q - P;  /* value is 2      */
                };
```

5.21 Shift operators

```
expression -> expression << expression
           -> expression >> expression
```

The value of E1<<E2 is E1 left shifted E2 bits. The value of E1>>E2 is E1 right shifted E2 bits; if E1 is a signed type, the right shift will be arithmetic (sign fill), otherwise it will be logical (zero fill). Both operands must be scalars. The type of the result is LONG if E1 is LONG, INT if E1 is INT, and otherwise UNSIGNED. The value is undefined unless the value of E2 is less than the number of bits in the result. The shift operators group left to right.

5.22 Bit operators

```
expression -> expression && expression
           -> expression || expression
           -> expression !! expression
```

The && operator performs a bitwise AND function of its operands; the || operator performs a bitwise inclusive OR function; the !! operator performs a bitwise exclusive OR function. The && operator has higher precedence than || and !!. Both operands must be scalars. The conversions described in #5.2 occur. The bit operators group left to right.

5.23 Relational operators

```
expression -> expression =  expression
           -> expression ~= expression
           -> expression <  expression
           -> expression >  expression
           -> expression <= expression
           -> expression >= expression
```

The relational operators are = (equal), ~= (not equal), < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal). They yield a 1 result if the specified relation is true and a 0 if it is false; the type of the result is UNSIGNED. The two operands must be of compatible types. If either operand is a signed scalar type, the comparison is signed; if both operands are unsigned scalars, the comparison is unsigned. Pointer comparisons are unsigned. Arrays, structures, and unions are compared one byte at a time, from lowest address to highest, using unsigned arithmetic. The relational operators group left to right, but A<B<C will compare a 1 or 0 with C.

5.24 Logical operators

```
expression -> expression & expression
           -> expression | expression
```

The binary & operator results in 1 if both operands are nonzero, otherwise 0. The | operator results in 1 if either operand is nonzero, otherwise 0. The & operator has a higher precedence than |. The first operand is always evaluated first; moreover the second operand is not evaluated unless needed to determine the result. The operands need not be the same type, but each must be a scalar or pointer. The result is always UNSIGNED. The logical operators group left to right.

5.25 Conditional operator

```
expression -> expression ? expression : expression
```

If the value of the first operand is nonzero, the result is the value of the second operand, otherwise the result is the value of the third operand. Only one of the second and third operands is evaluated. The first operand must be a scalar or pointer. If the second and third operands are scalars, the type of the result is given by #5.2, otherwise they must be compatible pointers and the result is the same type. The conditional operator groups right to left.

5.26 Assignment operators

```
expression -> expression  := expression
           -> expression  *= expression
           -> expression  /= expression
           -> expression  %= expression
           -> expression  += expression
           -> expression  -= expression
           -> expression <<= expression
           -> expression >>= expression
           -> expression &&= expression
           -> expression ||= expression
           -> expression !!= expression
```

The simple assignment operator := places the value of the second operand in the variable referred to by the left operand, which must be an lvalue. The other assignment operators, of the form E1op=E2, are equivalent to (E1):=(E1)op(E2); however, E1 is only evaluated once. For := the operands must be compatible types. For the other assignment operators, the right operand must be a scalar; the left operand must be a scalar, or for += and -= it may be a pointer, in which case the pointer arithmetic is as in #5.20. The result of an assignment expression has the value and type of the left operand after the assignment, but is not an lvalue. The assignment operators group right to left.

```
Examples:  VAR I, J, K INT;
               PI ^ INT;
           {   I := 1;
               J := K := 0;
               PI := 0;
               };
```

6. Statements

Statements are normally executed in sequence, but several kinds of statements alter the flow of control.

6.1 Expression statement

statement -> expression

An expression may be used as a statement. The last operator performed by the expression must be one that has a side effect, namely function call, ++, --, assignment, or comma. If a function call which returns a value is used as a statement, the value is ignored.

6.2 Compound statement and block

```
statement -> VAR declaration-list { statement-list_opt }
          -> { statement-list_opt }

declaration-list -> declaration ;
                 -> declaration ; declaration-list
```

```
statement-list -> statement ;
               -> statement ; statement-list
```

A compound statement permits several statements to be used where one is expected. If the compound statement starts with declarations (#7.), it is a local block. The only function declarations permitted in a local block are EXTERN declarations. Initializations of local variables occur only when the program is loaded, not when the block is entered.

Note that each statement in a compound statement must be followed by a semicolon.

```
Examples:  {   I := J;
               J++;
               };
           VAR M INT;
               P ^ INT;
               EXTERN F ();
           {   M := P^;
               F();
               };
```

6.3 Conditional statement

```
statement -> IF expression THEN statement
          -> IF expression THEN statement ELSE statement
```

The scalar or pointer expression is evaluated and if it is nonzero, the statement following the THEN is executed. If the expression is zero and there is an ELSE statement, that statement is executed. There is never a semicolon preceding an ELSE. The ELSE is associated with the nearest possible IF in ambiguous cases such as

```
           IF e THEN IF e THEN s ELSE s
```

6.4 Loop statements

```
statement -> LOOP statement

statement -> WHILE expression
```

The LOOP statement causes the substatement to be repeated. If the statement following LOOP is a compound statement, one or more of its substatements may be a WHILE statement. Execution of a WHILE statement terminates execution of the LOOP if the value of the scalar or pointer expression is zero.

```
Examples:  LOOP SUBR();                     /* loops forever      */
           I := 0;
           LOOP WHILE ++I < 10000;          /* loops 10000 times */
           I := 0;
           LOOP {
              A[I] := 0;       /* clears A[0] through A[99] */
           WHILE I++ < 99;
              };
```

6.5 CASE statement

```
statement -> CASE expression { case-statement-list_opt }

case-statement-list -> case-statement ;
                    -> case-statement ; case-statement-list

case-statement -> case-label-list : statement

case-label-list -> case-label
                -> case-label , case-label-list

case-label -> constant-expression
           -> DEFAULT
```

The case statement causes control to be transferred to one of several statements depending on the value of an expression. The expression must be a scalar. When the CASE statement is executed, control is transferred to the statement preceded by a constant expression having the same value as the expression. If no statement is preceded by a matching value, control is transferred to the statement preceded by a DEFAULT if there is one, otherwise control is transferred to the next statement. After execution of one of the statements in the CASE, control is transferred to the next statement after the CASE statement.

```
Example:   CASE I {
              1:       X++;
              2, 3:    X--;
              DEFAULT: X := 0;
              };
```

6.6 GOTO statement

```
statement -> GOTO identifier
```

Control may be transferred unconditionally by this statement.

The identifier must be a label (#6.7) on a statement in the same function.

6.7 Labeled statement

```
statement -> identifier : statement
```

Any statement may be preceded by a label. The colon declares the identifier as a label. The label may be used in a GOTO. The scope of the label is the function in which it appears (see scope rules #9.).

6.8 Null statement

```
statement ->
```

A null statement causes no special action but may be used wherever any other statement may appear.

7. Declarations

```
program ->
        -> declaration ; program

class -> ENTRY
      -> EXTERN
      -> STATIC
```

A program module is a sequence of declarations of types, variables, and functions. Variable and type declarations and external function declarations may also appear in blocks (#6.2). Optional class specifications control the scope and storage allocation of declared objects.

7.1 Type declaration

```
declaration -> TYPDEF identifier-list type
```

The TYPDEF declaration declares one or more identifiers to be an equivalent name for the specified type; such an identifier may be then used anywhere a type is required.

```
Example:   TYPDEF ID [8] CHAR;
           A ID;
           P ^ ID;
```

7.2 Variable declaration

```
declaration -> class_opt identifier-list type_opt
            -> class_opt identifier type_opt := initializer

initializer -> expression
            -> { initializer-list_opt }

initializer-list -> initializer
                 -> initializer , initializer-list
```

A variable declaration defines one or more variables of the specified type. The type may be any type but a function type.

The class ENTRY in a global variable declaration causes the variable name to be an entry point which may be referenced by other program modules when the modules are linked together by the LINK utility. The class EXTERN causes no storage to be allocated but declares that the name is defined as an entry point in some program module, such as with the ENTRY class in another DASL program module.

If a variable has been declared EXTERN, such as by an include file, it may be redeclared as ENTRY in the same block; the second declaration omits the type specification. In all other cases the type is required.

The class STATIC causes a variable declared in a block inside a function to be permanently allocated instead of being allocated dynamically each time the function is called. Variables declared globally are always STATIC.

A variable which is global or STATIC may be given an initial value with an initializer. If an initializer is specified only one identifier may be given in the identifier list. An initializer for a scalar or pointer variable is an expression of compatible type; an array of characters may also be initialized by a string constant of identical size.

An array or STRUCT variable may be initialized by specifying, inside braces, a list of initializers for the components of the aggregate variable in increasing order; the initializers for components of a character array may include string constants. Fewer initializers may be specified than the number of components if not all are to be initialized. Braces may be nested if the component of an array is also an array or structure, or a component variable of a structure is an array or named structure. If an array is initialized, the first bound of the array type specification may be omitted, and the compiler will fill in the bound from the number of initializers specified. A UNION is treated like a STRUCT, but only the first alternative of the UNION is initialized.

Initializer expressions must be values which are constants when the program is linked; such expressions may involve numeric and string constants, addresses of static variables, and addresses of arrays subscripted by constants.

```
Examples:  ENTRY I INT := 0;
           PC ^ CHAR := &'1';
           A [] CHAR := '123';
           AA [] [2] INT := { { 1, 2 }, { 3 }, { 4 } };
           AAA [10] CHAR := { '123', 015 };
           S STRUCT { A INT; UNION { B, C INT; }; D ^ CHAR; } :=
                    { 1,              2,             &AAA[0]+1 };
```

7.3 Function declaration

declaration -> fclass identifier-list type
            -> fclass identifier type$_{opt}$:= statement

fclass -> class$_{opt}$ RECURSIVE$_{opt}$

A function declaration declares one or more identifiers to be names of functions and may specify the code to be executed when the function is called. The type must be a function type. With the exception of EXTERN declarations, all function declarations occur in the outer block; function declarations may not be nested.

The ENTRY and EXTERN class specifications function the same as for variable declarations (#7.2). The RECURSIVE class declares that the function may be called recursively.

If the statement body is present, only one identifier may be specified. The statement body is omitted if the class is EXTERN, or if the function type needs to be declared before the function body is given, as in cases of mutual recursion. In this second case the function name must be redeclared with the body provided. A function name which has been declared as EXTERN may also be redeclared, as ENTRY. In these two cases, the type is omitted on the second declaration, but the ENTRY and RECURSIVE classes, if needed, are specified on the second declaration. In all other cases the type must be specified.

The statement becomes a block (#9.) which is executed when the function is called. The formal parameters in the function type become local variables in this block; they are initialized to the values passed as the actual parameters. Within this block, the variable name RESULT refers to the value of the function; an assignment to this variable sets the value returned by the function.

```
Examples:  EXTERN F (X INT);

           G (X INT) INT;

           H () INT :=
           {   RESULT := G(1);
               };

           ENTRY G :=
           VAR I INT;
           {   IF H() THEN I := G(X);
               F(I);
               RESULT := I;
               };
```

8. Macros

Macros provide a text substitution facility. In the simplest case, occurrences of a particular identifier may be replaced by any desired string of characters; this permits constants to be given a name, increasing program clarity. Macros may also have parameters, providing the equivalent of function calls without the function call overhead.

Five macro definitions are built in to the compiler. These provide the ability to define a new macro, compare two text strings, increment a numeric text string, select a substring, and include additional files in a source program.

Macros are independent of the other syntax rules of the language, and a macro call may occur anywhere except inside a comment or a quoted string constant.

8.1 Macro call

A macro is called by one of the sequences

```
identifier    or    identifier(parameter,parameter,...)
```

where the identifier names a predefined macro or a macro which has been defined by the DEFINE macro (#8.3). In the first case the macro is called with no parameters; in the second case there are zero to nine parameters to the macro, each of which is a string (sequence) of characters. Blanks are significant inside parameters; line ends are treated as characters. If a parameter contains a (character, then any commas until the balancing) character are considered part of the parameter and not a parameter separator (see also evaluation deferral #8.2). Commas and parentheses inside comments and quoted string constants are not treated as macro delimiters.

When a macro call appears, each parameter is scanned, with any macro calls within the parameter being performed, and the parameters are saved. Next, the parameters are substituted into the macro definition according to the parameter specifications in the macro definition. Finally the adjusted macro definition replaces the macro call, and the result is rescanned in case it contains more macro calls. If a nested macro call is preceded by the # symbol, this rescanning does not occur.

If a macro call supplies fewer parameters than the macro definition requires, the unspecified parameters are considered to be null strings. If the call supplies extra parameters, they are ignored.

```
Examples:  X
           F(1+2,2)
```

8.2 Evaluation suppression

The character pair #[indicates that the following characters up to a balancing #] are all to be treated as ordinary characters; macro names, parentheses, and commas have no special meaning. The #[and #] symbols are stripped. If these symbols occur in a macro parameter, any suppressed macro calls may later occur if the suppressed string is rescanned after substitution in the macro definition. The #[and #] symbols may be nested, with one level being removed each time they are scanned. The #[and #] symbols are not treated as macro delimiters inside comments and quoted string constants.

```
Example:   #[ X #]
```

8.3 Macro definition

A macro is defined by the macro call

```
DEFINE(identifier,definition)
```

which is replaced by a null string but has the side effect of defining the identifier as the name by which the macro is to be called and the definition as the string to be substituted for the macro call. Within the definition, the character pair #n, where n is a digit from 1 to 9, will be replaced by parameter n when the macro is called, even inside comments, quoted string constants, and #[#] symbols. The definition may need to be surrounded with the evaluation suppression symbols #[and #] if it contains macro calls which should not occur until the macro being defined is called.

The identifier will normally be defined as a local definition in the current block, and the definition will be forgotten at the end of the block. If the identifier is already defined as a macro, however, that definition (even if not in the current block) will be replaced with the new definition and no error will occur. In this case, the identifier should be surrounded by the #[#] symbols in the DEFINE so the old definition is not substituted when the parameters to the DEFINE are scanned.

```
Examples:  DEFINE(X,1)
           DEFINE(SQR,((#1)*(#1)))
           DEFINE(NEW, <^CHAR>(#1) := ALLOC(SIZEOF (#1)^ )
           DEFINE(GETC, ( LP^=EOL ? (GETLINE(),LP++^) : LP++^ ) )
```

Note that no semicolon follows a DEFINE. Note also that it is often good practice to surround a macro parameter reference with parentheses to force proper expression evaluation order.

8.4 String comparison

The macro call

```
           IFELSE(string1,string2,equal,unequal)
```

compares the first two character strings. If they are equal, the result of the macro call is the third string; otherwise the result is the fourth string. If the two strings are unequal length, they are considered unequal. If the third or fourth strings contain macro calls, it may be necessary to surround them with the evaluation suppression symbols #[#] to defer the macro calls when the IFELSE parameters are first scanned.

```
Examples:  IFELSE(1,2,3,4)     is replaced by 4
           IFELSE(1,1,3,4)     is replaced by 3
           IFELSE( ,,3,4)      is replaced by 4
```

8.5 Numeric incrementation

The macro call

```
           INCR(number)
```

gives as a result a string which is the decimal number following the value of the parameter. The parameter may be an unsigned decimal, octal, or hexadecimal number (#2.3), or it may be a string constant (#2.4), in which case the value of the first character of the string is used. If the parameter is neither of these the value is considered to be zero. The parameter may have leading blanks.

Example:

```
DEFINE(E,#[IFELSE(#2,,,#[DEFINE(#2,#1)E(INCR(#1),#3,#4,#5,#6)#])#])
E(0,A,B,C)
```

This complex example is a recursive macro which defines a list of identifiers to be ascending numerical values. When the macro call E(0,A,B,C) is scanned, it is replaced by the definition of E with the parameters filled in. The result is

```
           IFELSE(A,,,#[DEFINE(A,0)E(INCR(0),B,C,,)#])
```

which is then rescanned. The parameters to the IFELSE macro are scanned, but the #[#] symbols inhibit the macro calls in the fourth parameter. Since the first two parameters are not equal, the result is

```
           DEFINE(A,0)E(INCR(0),B,C,,)
```

which is then rescanned. The DEFINE call defines A to be the string 0 and is replaced by a null string. Next the E macro is seen and its parameters are scanned; the INCR(0) macro call is replaced by the string 1. Next the call to E is replaced by the definition of E

```
           IFELSE(B,,,#[DEFINE(B,1)E(INCR(1),D,,,)#])
```

and everything repeats again but B is defined instead of A. The third time C is defined, and the fourth time the recursion terminates with the processing of

```
           IFELSE(,,,#[DEFINE(,3)E(INCR(3),,,,)#])
```

in which the first two parameters of the IFELSE are equal, so the IFELSE call is replaced with the third parameter, which is null.

8.6 Substring selection

The macro call

```
            SUBSTR(string,start,length)
```

selects the portion of the string parameter specified by the start and length parameters. The start and length parameters are numbers with the same rules as for the INCR macro. The result is the number of characters specified by the length parameter, starting at the position specified by the start parameter (counting from 0). If the length parameter is omitted, the result is the rest of the string from the start position. If the string is shorter than required by the specified start and length, the result is the portion of the string within the specified start and length.

```
Examples:   SUBSTR(ABC,1)      is replaced by     BC
            SUBSTR(ABC,0,2)    is replaced by     AB
            SUBSTR(ABC,2,2)    is replaced by     C
```

8.8 File inclusion

The macro call

```
            INCLUDE(filespec)
```

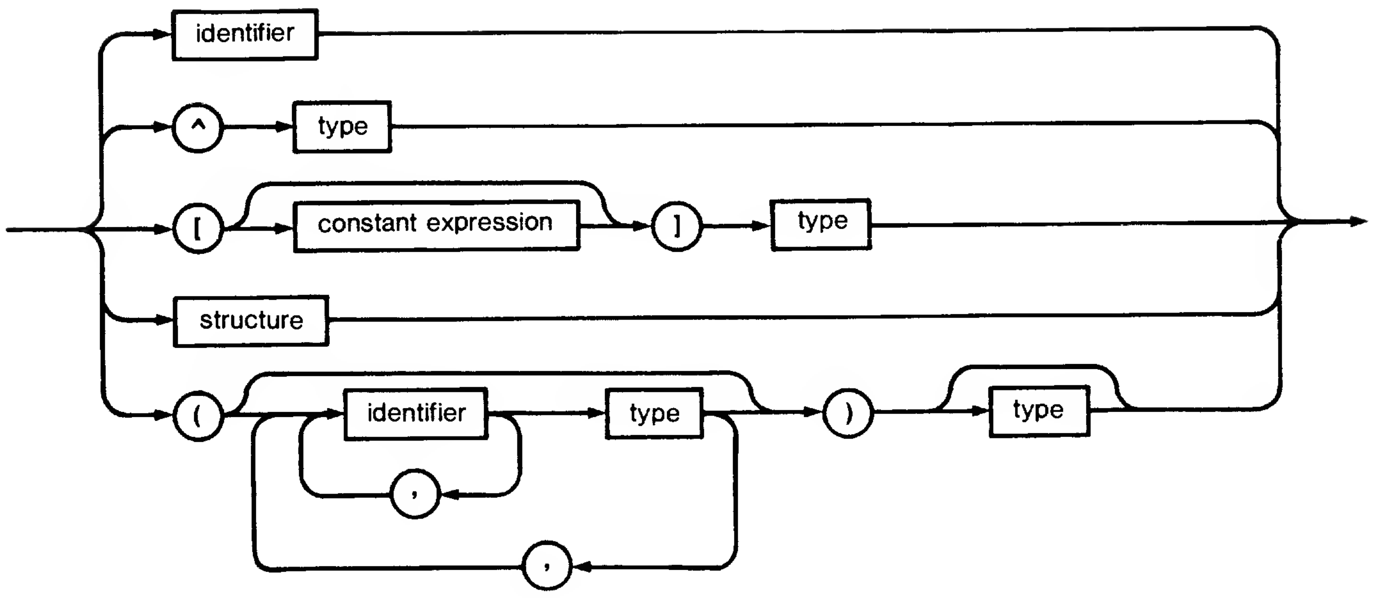
is replaced by a null string but as a side effect causes subsequent input lines to come from the specified file. Includes may be nested.

```
Example:    INCLUDE(COMMON)
```

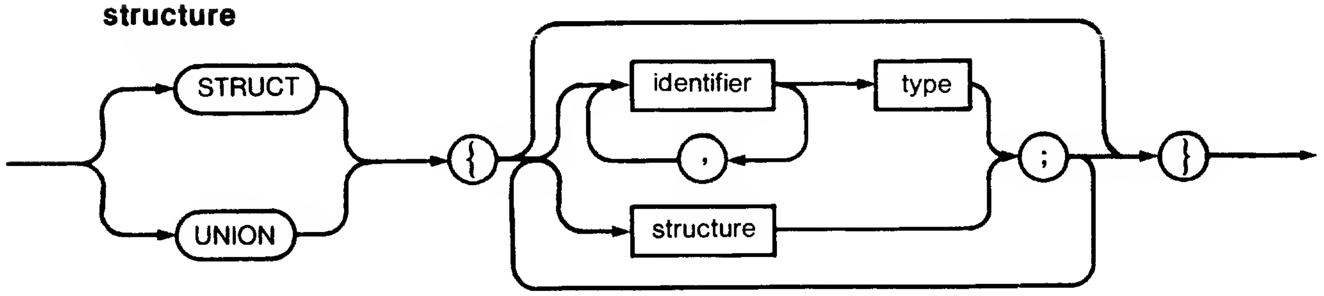
9. Scope rules

An identifier is declared by a TYPDEF, variable, or function declaration, by its use as a formal parameter, by its use as a label, or by its definition as a macro. Every identifier is declared in a particular block; the whole input file is considered to be the global block. Except for a macro name, an identifier may have only one declaration in a particular block, but a declaration may duplicate an identifier which has been declared in an enclosing block; the outer definition is then temporarily hidden. The definition of an identifier is visible in the block in which it is declared and any inner blocks in which it is not hidden by a redeclaration. All identifiers must be declared before they are used, except for labels and the types to which pointers point.

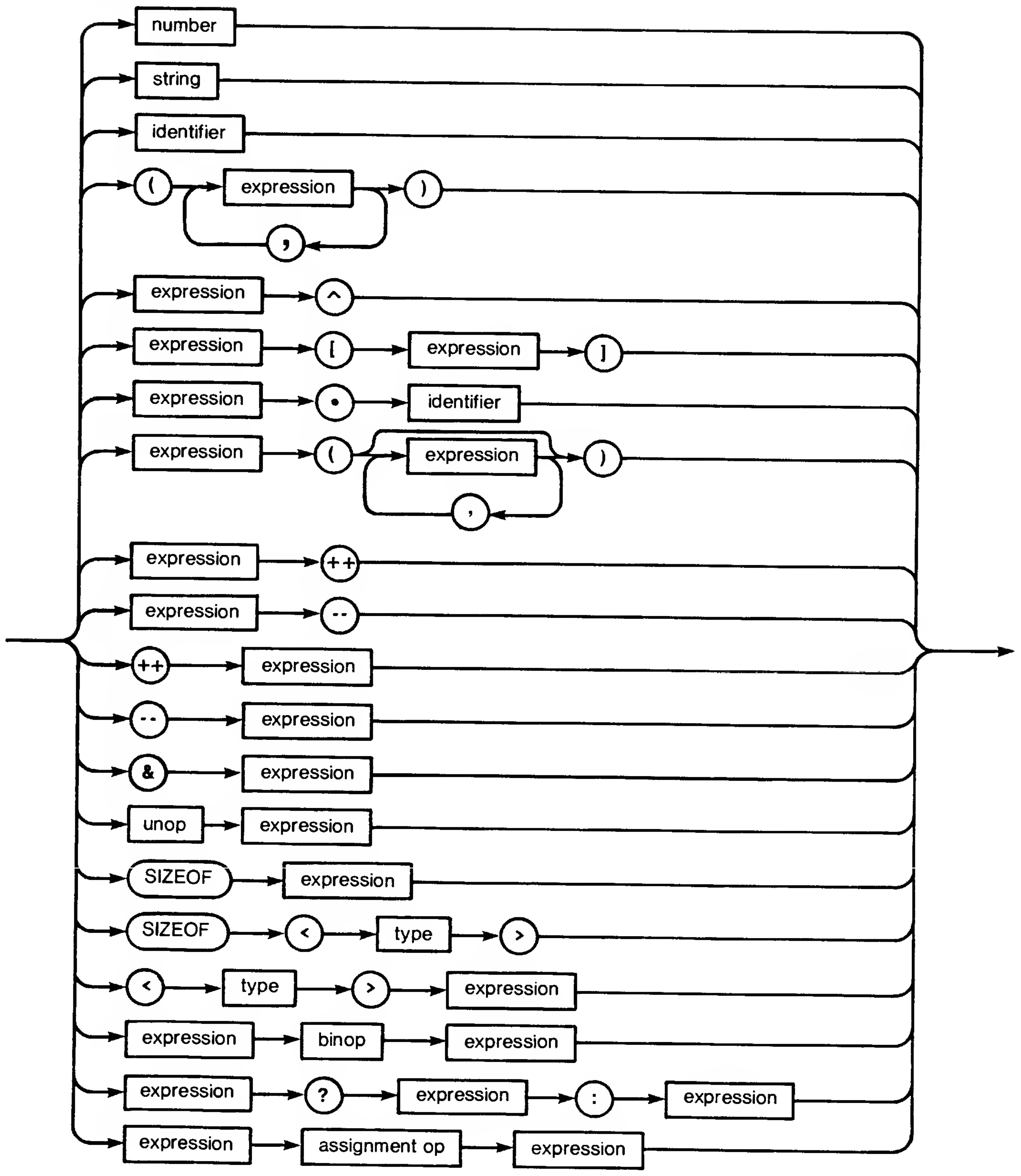
type



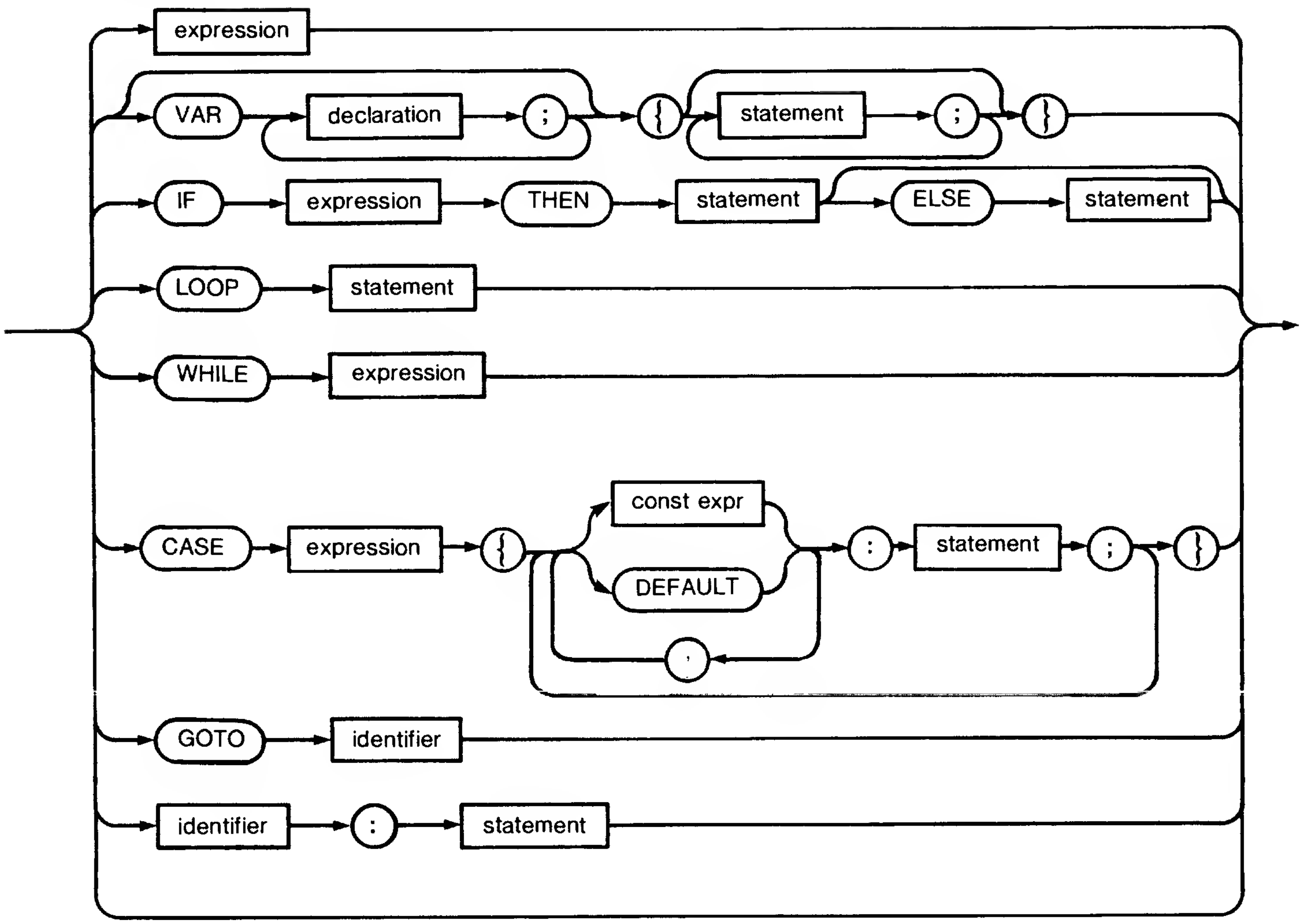
structure



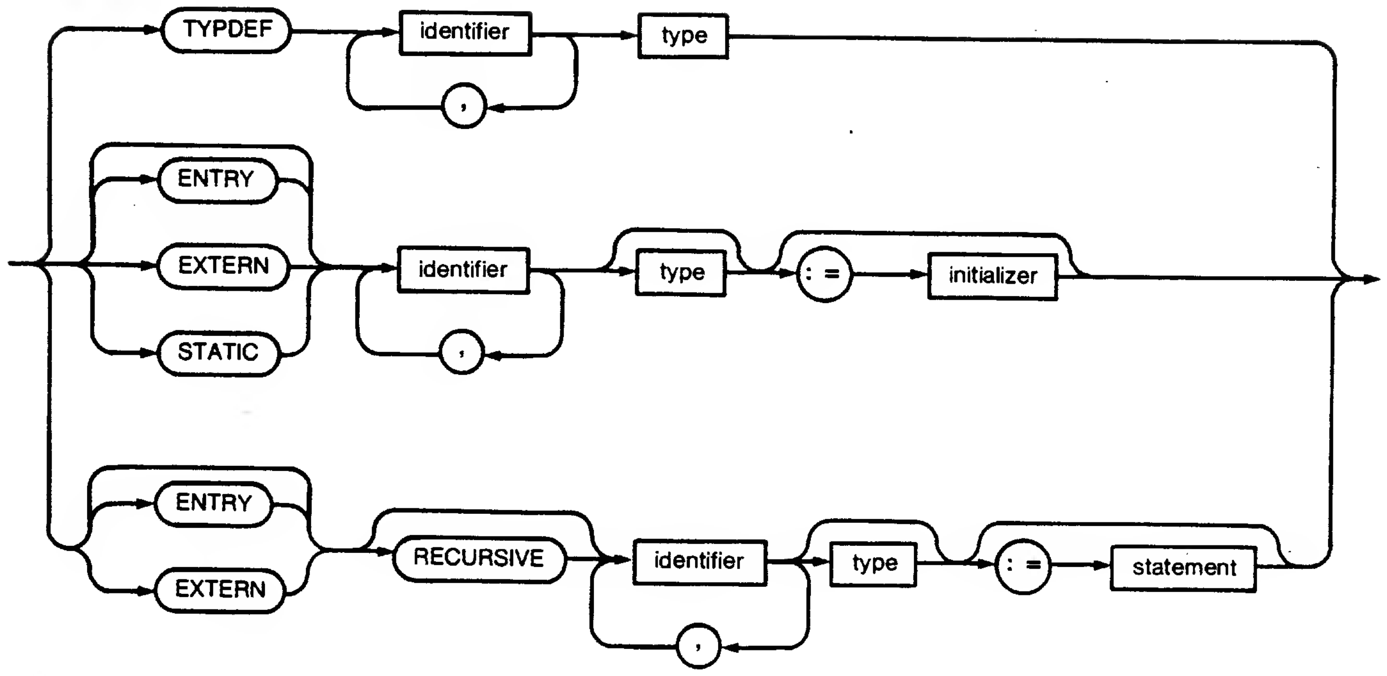
expression


number
string
identifier
(
expression
,
)
expression
^
expression
[
expression
]
expression
•
identifier
expression
(
expression
,
)
expression
++
expression
--
++
expression
--
expression
&
expression
unop
expression
SIZEOF
expression
SIZEOF
<
type
>
<
type
>
expression
expression
binop
expression
expression
?
expression
:
expression
expression
assignment op
expression

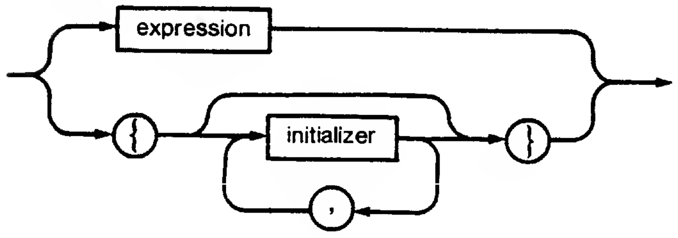
statement


expression
VAR
declaration
;
{
statement
;
}
IF
expression
THEN
statement
ELSE
statement
LOOP
statement
WHILE
expression
CASE
expression
{
const expr
DEFAULT
,
:
statement
;
}
GOTO
identifier
identifier
:
statement

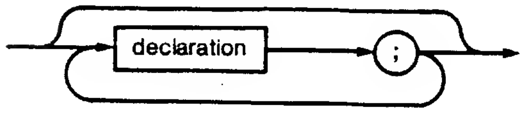
declaration



initializer



program



Appendix 2. DASL Calling Sequences

This appendix describes the assembly language code generated by the DASL compiler for function calls and for recursive function storage allocation on the processors having the 5500 or 6600 instruction set. This information is useful for programmers writing assembly language routines to be called from DASL, and for programmers wishing to control the storage management for recursive functions.

Function Calling Sequences

The function calling sequence is independent of whether the called function is recursive or not. There are two cases, however, depending on the number of parameters to be passed.

If there are three parameters or less, the third parameter (if any) is passed in the BC register pair, the second in DE, and the first in HL.

If there are more than three parameters, they are pushed on the stack in reverse order, with the first parameter being pushed last.

In all cases, if the formal parameter type is one byte long, the high byte of the register pair or stack entry is undefined. If the formal parameter type is LONG, the address of a memory location containing the LONG value is passed.

If the function has a result, it is passed in the A register if a single byte, or in HL if two bytes. If, however, the result type is LONG, the code generated is as if the function had no result but had another parameter of type pointer to LONG; the generated code passes the address of a location in which to place the LONG result.

```
Example:   EXTERN F (I INT, B BYTE);
           EXTERN G (A, B, C, D BYTE) BYTE;

           F(1, 2);
                                              LE     2
                                              HL     1
                                              CALL   F
```

```
          IF G(1, 2, 3, 4) THEN ...
                                             PUSH  4
                                             PUSH  3
                                             PUSH  2
                                             PUSH  1
                                             CALL  G
                                             ORA
                                             JTZ   ...
```

Recursive Function Storage Allocation

The DASL run time library D$LIB includes a default set of routines for managing recursive function calls. Many programs with recursive functions will be able to use these routines. They allocate recursive function storage from a fixed area in the library. If this default area is not of adequate size, the programmer may include in his program a storage area definition such as

```
          ENTRY D$STACK [3000] BYTE; ENTRY D$STACKE [0] BYTE;
```

defining an area of 3000 bytes.

The default routines will call a function named D$STKOF if the default or user supplied area is exceeded. There is a default routine in the run time which outputs an error message, but the user may supply his own error routine, such as

```
          ENTRY D$STKOF () := $ERROR();
```

which simply calls the RMS error exit. Such an error routine must not return normally to its caller.

The user may also perform his own storage management. At entry to a recursive function, the compiler generates the following code if there are three parameters or less

```
          PUSH BC                       (if 3 parameters)
          PUSH DE                       (if 2 or more parameters)
          PUSH HL                       (if 1 or more parameters)
          LC   <number of parameters>
          DE   <number of bytes needed>
          CALL D$RENREG
```

If there are more than three parameters, the code generated is

```
          LC   <number of parameters>
          DE   <number of bytes needed>
          CALL D$RENSTK
```

The D$RENREG or D$RENSTK routine should allocate a storage area of the number of bytes specified in DE, leave its address in the ENTRY variable D$RFRAME, and pop the parameters and function return address off the stack and store them in the storage area.

The format of the local storage area is

```
      ---------------------------------------------
    0 | available for link to previous stack frame |
      |--------------------------------------------|
    2 | available for return address               |
      |--------------------------------------------|
    4 | first parameter                            |
      |--------------------------------------------|
      |                                            |

      |                                            |
      |--------------------------------------------|
      | last parameter                             |
      |--------------------------------------------|
      |                                            |

      |                                            |
      ---------------------------------------------
 (DE)
```

At the end of a recursive function, the code generated is

```
        JMP   D$REXIT
```

This routine should deallocate the local storage, restore D$RFRAME to point to the previous stack frame, and return to the caller of the function. The A and HL registers should be preserved, since they may contain a returned function value.

Appendix 3. The D$INC INCLUDE

The D$INC INCLUDE file provides standard definitions for DASL programs. This appendix describes the use of this file.

Several symbolic constants are defined: MAXINT (077777, maximum value of an INT), MAXUNSIGNED (0177777), MAXLONG (017777777777), NIL (0, null pointer), and FALSE (0) and TRUE(1).

The macro ENUM is defined as BYTE, and additionally defines up to nine arguments as ascending values from zero, as in the variable declaration

```
      STATUS ENUM(NORMAL, EOF, ERROR);
```

The macro ENUMV is similar, but just defines its arguments to be ascending values from the first argument, as in

```
      ENUMV(1, CLOSED, FORWARD, BACKWARD)
```

The macros SET and SETV are similar, but define their arguments as successive powers of two. The macro SETW is like SET but is defined as UNSIGNED.

The type ULONG is a definition for 24 bit values. The routines D$GET24 and D$PUT24 convert between 24 bit ULONG and 32 bit LONG values.

The external function D$MOVE performs a block move of from 0 to 65535 bytes, from the BYTE address given by the first parameter to the BYTE address given by the second parameter, for the number of bytes given by the third parameter. The function D$MOVER is similar but takes the ending addresses of the blocks. The external function D$COMP performs a block compare between two strings given by the first two parameters, which are BYTE pointers, for the number of bytes given by the third parameter. If the two strings are equal, the function returns zero, otherwise it returns the difference of the first two differing bytes (byte in first string minus byte in second string).

The external function D$SC will perform an RMS system call; its argument is the system call number. The function gets the register values passed into the system call from external variables corresponding to the registers; these variables may be assigned values by the program before the D$SC call:

```
    D$X, D$A, D$B, D$C, D$D, D$E, D$H, D$L, D$XA, D$BC, D$DE, D$HL
```

The D$SC function returns the register values after the system call in these same variables; in addition the condition code returned by the system is returned as the function value and in the external variable D$CC with a bit for each of the possible flags. The masks for these bits are

    D$CFLAG, D$ZFLAG, D$SFLAG, D$PFLAG

The external function D$CALL works like D$SC, but performs a CALL to an arbitrary subroutine; the argument is the subroutine address, of type pointer to D$CALLF.

Several external 256 byte arrays, D$BUF1 through D$BUF5, are defined. These arrays are aligned on memory page boundaries so they may be used as RMS I/O buffers.

```
/* DASL STANDARD INCLUDE - SEPTEMBER 15, 1982 */

DEFINE(MAXINT,077777)
DEFINE(MAXUNSIGNED,0177777)
DEFINE(MAXLONG,017777777777)
DEFINE(NIL,0)

DEFINE(ENUMV,#[IFELSE(#2,,,
        #[DEFINE(#2,#1)ENUMV(INCR(#1),#3,#4,#5,#6,#7,#8,#9)#])#])
DEFINE(ENUM,#[DEFINE(#1,0)ENUMV(1,#2,#3,#4,#5,#6,#7,#8,#9)#]BYTE)

DEFINE(SETV,#[IFELSE(#2,,,
        #[DEFINE(#2,(#1))SETV(#1*2,#3,#4,#5,#6,#7,#8,#9)#])#])
DEFINE(SET,#[SETV(1,#1,#2,#3,#4,#5,#6,#7,#8)#]BYTE)
DEFINE(SETW,#[DEFINE(#1,1)SETV(2,#2,#3,#4,#5,#6,#7,#8,#9)#]UNSIGNED)

ENUMV(0,FALSE,TRUE)

TYPDEF ULONG, ILONG STRUCT {
   LSW UNSIGNED;
   MSB BYTE;
   };

EXTERN D$GET24(pul ^ ULONG) LONG;
EXTERN D$PUT24(l LONG, pul ^ ULONG);

EXTERN D$MOVE, D$MOVER (S, D ^ BYTE, N UNSIGNED);
EXTERN D$COMP (S, D ^ BYTE, N UNSIGNED) INT;

EXTERN RASLRES$ ();

/* SYSTEM INTERFACE */

TYPDEF D$CCODE SET(D$CFLAG,D$ZFLAG,D$SFLAG,D$PFLAG);

TYPDEF D$CALLF ();

/* Important: reference to the following register variables, D$SC,
   D$CALL, and D$CC should be avoided outside of macro definitions
   in include files to preserve machine independence.  Reference
   to the above D$xFLAG names is okay.
*/

EXTERN D$X, D$A, D$B, D$C, D$D, D$E, D$H, D$L BYTE;
EXTERN D$XA, D$BC, D$DE, D$HL UNSIGNED;
EXTERN D$SC (SCNUM BYTE) D$CCODE;
EXTERN D$CALL (F ^ D$CALLF) D$CCODE;
EXTERN D$CC D$CCODE;

EXTERN D$BUF1, D$BUF2, D$BUF3, D$BUF4, D$BUF5 [256] BYTE;
```

Appendix 4. The D$RMS Include

D$RMS is the standard DASL include file which contains definitions required to deal effectively with the RMS operating environment. In particular, SIO and other RMS I/O packages rely heavily on the availability of these definitions.

```
/* DASL COMMON RMS DEFINITIONS - DECEMBER 21, 1982  */

/*

   This file contains definitions for commonly used RMS nucleus
and UFR functions.  The hope is that most programs will just need
these definitions, the ones in D$INC, and the definitions for
whatever I/O package is being used.  Additional INCLUDE files are
organized according to chapters in the nucleus and UFR user's
guides as follows:

   D$RMSGEN     Chapter      3.  General System Functions
   D$RMSPROG    Chapter      4.  Program Loading and Execution
   D$RMSMEM     Chapter      5.  Memory Management
   D$RMSTASK    Chapter      6.  User Multi-tasking
   D$RMSIO      Chapters 7-10.  File Handling, Block I/O, Disk,
                                  Printer, Pipe
   D$RMSSTRUCT Chapter      8.  Disk Structure
   D$RMSWS      Chapter     11.  Workstation
   D$RMSSPEC                    Special System Calls

   D$UFRENV     Chapter      2.  Environment Handling
   D$UFRERR     Chapter      3.  Error Handling
   D$UFRNUM     Chapter      4.  Numeric Manipulation
   D$UFRSYS     Chapter      5.  System Interface
   D$UFRSCAN    Chapter      6.  Command Interpreter (Scanning)
   D$UFRWS      Chapter      7.  Workstation Interface
   D$UFRGEN     Chapter      8.  General Utility
   D$UFRLIB     Chapter      9.  Library Manipulation
   D$UFRNQDQ    Chapter     10.  NQ/DQ
   D$UFRRLD     Chapter     11.  Relocating Loader
   D$UFRMEM     Chapter     12.  Memory Management
   D$UFRWFIO    Chapter     13.  Work File I/O

   D$FAR                        File Access Routines

   D$ERRNUM                     System Error Numbers

   D$PCR                        Program Communications Region

*/
```

```
/* NUCLEUS DEFINITIONS */

DEFINE($NOADR,0177777)
DEFINE($NOPSK,0377)

DEFINE($LMCV,0371)
ENUMV(0371,$LSPC,$LEOR,$LEOF,$LST,$LEOB,$LXX,$LDEL)

.       The $ERRC structured variable contains the RMS error code
. after an error return from a system call or UFR.  The system
. code number or UFR class is in $ERRC.$FUNC (symbolic values are
. in the D$ERRNUM include).  The error number is in $ERRC.$CODE.
. This structure should be used instead of the machine dependent
. places D$B and D$C.

TYPDEF $ERRCODE STRUCT {
   $CODE, $FUNC BYTE;
   };
EXTERN $ERRC $ERRCODE;

TYPDEF $NAMET [12] CHAR;
TYPDEF $EXTT [4] CHAR;
TYPDEF $NAMEEXT STRUCT {
   $NAME $NAMET;
   $EXT  $EXTT;
   };
TYPDEF $HSI [32] CHAR;
TYPDEF $PACKPW [6] BYTE;
TYPDEF $LSN ULONG;
TYPDEF $FILEPTR UNION {
   STRUCT {
      $FPTRBUFOF BYTE;
      $FPTRLSN $LSN;
      };
   $FP LONG;
   };
TYPDEF $LNAMET [8] CHAR;
TYPDEF $TIME [5] BYTE;
```

```
/* CHAPTER 7.  GENERAL FILE HANDLING */

TYPDEF $PFDBBUF STRUCT {
   PSK, PAGE BYTE;
   SYS UNSIGNED;
   };

TYPDEF $PFDB STRUCT {
   $PFVID UNSIGNED;
   UNION {
      $PCLSN $LSN;                           /* DISK */
      STRUCT {
         UNION {
            $PBLKL UNSIGNED;                 /* PIPE AND TAPE */
            STRUCT {
               $PSYSCODE BYTE;               /* DIRECT RIM */
               $PTASKID BYTE;                /* DIRECT RIM */
               };
            };
         UNION {
            $PTIMER BYTE;                    /* PIPE AND DIRECT RIM */
            $PSUBF BYTE;                     /* TAPE */
            };
         };
      };
   $PTODO BYTE;
   $PDONE BYTE;
   $PMXBF BYTE;
   $PBUFL [1] $PFDBBUF;
   };

ENUMV(0,$OMREAD,$OMSHARE,$OMEXCL,$OMPREP,$OMCREAT,$OMCHECK,
 $OMREPAR,$OMBYPAS)

DEFINE($ENVTERM,0377)
DEFINE($MAXNPW,20)
```

```
TYPDEF $OPENPT STRUCT {
   $OTPFDB ^ $PFDB;
   $OTENV ^ CHAR;
   $OTFILE ^ $NAMEEXT;
   $OTKIND ENUM($DKWS,$DKDISK,$DKPIPE,$DKPRINT,$DKCASS,$DKMAGT,
       $DKCOMM,$DKTIMER);
      ENUMV(8,$DKCARDR,$DKCARDP,$DKPTR,$DKPTP,$DK883M,$DK863M,
       $DKSMPLR,$DKRIM)
      ENUMV(16,$DKFAX)
   $OTSUBK BYTE;
   $OTFEOFB BYTE;
   $OTFLEN $LSN;
   $OTFINC UNSIGNED;
   $OTFMT ENUM($FFMTSYS,$FFMTTMP,$FFMTTXT,$FFMTISM,$FFMTL55,
       $FFMTRAC,$FFMTDBC);
      ENUMV(7,$FFMTBAS,$FFMTMAC,$FFMTWPS,$FFMTJOB,$FFMTBIN,
       $FFMTUTX,$FFMTMFD)
      ENUMV(14,$FFMTUPF,$FFMTWPF,$FFMTAIM)
   $OTTIME $TIME;
   $OTSQL BYTE;
   $OTCODE BYTE;
   $OTONOS BYTE;
   $OTFID UNSIGNED;
   $OTRID BYTE;
   $OTX [2] BYTE;
   };

ENUMV(0,$CMUNCH,$CMSIZE,$CMCHOP,$CMKILL)

. EXTERN $CLOSE (mode BYTE, pfdb ^ $PFDB) D$CCODE;

/* CHAPTER 11.  WORKSTATION OPERATIONS */

ENUMV(015,$WSENTK)

ENUMV(0200,$EEOF,$EEOL)
ENUMV(0204,$WSBEEP,$WSCLICK)
ENUMV(0231,$ES)
ENUMV(0234,$H,$V)
ENUMV(0241,$HU,$HD)
```

```
/* CHAPTER 4.  PROGRAM LOADING AND EXECUTION CONTROL */

TYPDEF $STARTADR ^ D$CALLF;

. EXTERN $LOAD  (pfdb ^ $PFDB, lsn UNSIGNED, start ^ $STARTADR)
.  D$CCODE;

. EXTERN $EXIT  () D$CCODE;

. EXTERN $ERROR () D$CCODE;


/* UFR DEFINITIONS */

TYPDEF $ENVN [8] CHAR;
TYPDEF $NAMEEXTENV STRUCT {
   $NAME $NAMET;
   $EXT  $EXTT;
   $ENV  $ENVN;
   };

EXTERN $PCRCMDE $ENVN;
EXTERN $PCRCMDN $NAMEEXT;
EXTERN $PCRABTF SET($PCRAFGA);

/* CHAPTER 3.  ERROR HANDLING USER FUNCTION ROUTINES */

. EXTERN $MSGC  (errNum UNSIGNED, length ^ BYTE) D$CCODE;

DEFINE($MSGLGT,81)
EXTERN $MSG [$MSGLGT] CHAR;

. EXTERN $ERMSG ();

/* CHAPTER 5.  SYSTEM INTERFACE USER FUNCTION ROUTINES */

. EXTERN $OPEN (mode BYTE, openpt ^ $OPENPT) D$CCODE;
```

```
/* CHAPTER 6.  COMMAND INTERPRETER USER FUNCTION ROUTINES */

. Example usage of file name and option scanners:
.
.fileSpk [] $FILESPK := {
. {{'IN      ',<$NAMET>'',                   <$EXTT>'',     <$ENVN>''},
.  $FILNAMR,   $NOADR,                      &<$EXTT>'TEXT',&<$ENVN>''},
. {{'OUT     ',<$NAMET>'',                   <$EXTT>'',     <$ENVN>''},
.  $FILNDSP,   &fileSpk[0].$FSOSFT.$SFTNAM,&<$EXTT>'REL', &<$ENVN>''}
. };
.opt1 $OPTION := {'HELP    ',0,$OPTVCLR,0};
.opt2 $OPTION := {'CODE    ',0,$OPTVCLR,3}; opt2s [3] CHAR := '   ';
.optt $OPTTAIL := {$OPTTERM,0 };
.
.   IF $SCANOS(&opt1) && D$CFLAG THEN $ERMSG();
.   IF $SCANFLS(&fileSpk[0], 2) && D$CFLAG THEN $ERMSG();

TYPDEF $SFNT [8] CHAR;

TYPDEF $SFENT STRUCT {
   $SFTSFN $SFNT;
   $SFTNAM $NAMET;
   $SFTEXT $EXTT;
   $SFTENV $ENVN;
   };

TYPDEF $FILESPK STRUCT {
   $FSOSFT $SFENT;
   $FSOOPT SET($FILNAMR,$FILEXTR,$FILENVR,$FILANYR,$FILFDEF,
       $FILQMOK,$FILNDSP);
   $FSODNAM ^ $NAMET;
   $FSODEXT ^ $EXTT;
   $FSODENV ^ $ENVN;
   };

. EXTERN $SCANFLS (fileSpk ^ $FILESPK, num BYTE) D$CCODE;
```

```
TYPDEF $SONT [8] CHAR;

TYPDEF $OPTION STRUCT {
   $OPTSON $SONT;
   $OPTFLG SET($OPTFDEF,$OPTFVAL,$OPTFQOK,$OPTFABB,$OPTFSUB);
   $OPTVAL BYTE;  ENUMV(254,$OPTVSET,$OPTVCLR)
   $OPTMAX BYTE;
   $OPTSTR [0] CHAR;
   };

DEFINE($OPTTERM,0377)
TYPDEF $OPTTAIL STRUCT {
   $$OPTTERM BYTE;
   $OPTTOT BYTE;
   };

. EXTERN $SCANOS (option ^ $OPTION) D$CCODE;

/* CHAPTER 9.  LIBRARY MANIPULATION USER FUNCTION ROUTINES */

. EXTERN
.  $LBGTLSN (type BYTE, pfdb ^ $PFDB, name ^ $LNAMET, lsn ^ UNSIGNED)
.  D$CCODE;
```

```
DEFINE($CLOSE,(D$A:=#1,<^$PFDB>D$HL:=#2,D$SC(25)))

DEFINE($LOAD,(<^$PFDB>D$HL:=#1,D$DE:=#2,D$SC(45),
 (#3)^:=<$STARTADR>D$DE,D$CC))

DEFINE($EXIT,D$SC(42))

DEFINE($ERROR,D$SC(43))

EXTERN MSGC$ D$CALLF;
DEFINE($MSGC,(D$BC:=#1,D$CALL(&MSGC$),(#2)^:=D$A,D$CC))

EXTERN D$JUMP (F ^ D$CALLF);
EXTERN ERMSG$ D$CALLF;
DEFINE($ERMSG,D$JUMP(&ERMSG$))

EXTERN OPEN$ D$CALLF;
DEFINE($OPEN,(D$A:=#1,<^$OPENPT>D$HL:=#2,D$CALL(&OPEN$)))

EXTERN SCANFLS$ D$CALLF;
DEFINE($SCANFLS,(<^$FILESPK>D$HL:=#1,D$C:=#2,D$CALL(&SCANFLS$)))

EXTERN SCANOS$ D$CALLF;
DEFINE($SCANOS,(<^$OPTION>D$HL:=#1,D$CALL(&SCANOS$)))

EXTERN LBGTLSN$ D$CALLF;
DEFINE($LBGTLSN,(D$A:=#1,<^$PFDB>D$HL:=#2,<^$LNAMET>D$DE:=#3,
 D$CALL(&LBGTLSN$),(#4)^:=D$DE,D$CC))
```

*/